Writing FCode Programs for PCI

An Introduction to FCode Programming
Fully Compliant with *IEEE Standard 1275-1994*

Suite 115
480 San Antonio Road
Mountain View, CA 94040-1218
U.S.A.

Part No.: 000-0000-0000005-02
Revision C, August 1996

Contents

Tables

Figures

Preface

This manual, *Writing FCode Programs for PCI*, is derived from the Sun Microsystems manual *Writing FCode Programs* with adaptations specific to *IEEE Standard 1275-1994* and to PCI FCode drivers.

Who Should Use This Book

This manual is written for designers of PCI interface cards and other devices that use the FCode programming language. It assumes that you have some familiarity with PCI card design requirements and Forth programming.

This manual is oriented toward those developing FCode applications for PCI peripherals. However, most of the material applies to any FCode driver. The FCode language is defined by *IEEE Standard 1275-1994 Standard for Boot Firmware* (hereafter referred to as *Open Firmware*). The specifics of FCode for the PCI bus are defined in the *PCI Bus Binding to IEEE Standard 1275-1994 1*.6 (or later).

This manual also assumes that you have read and understood the *PCI Local Bus Specification, Revision 2.1* (or later).

How This Book Is Organized

- Chapter 1, “PCI Cards and FCode”, introduces the basic relationships between FCode device drivers and the hardware that they control.
- Chapter 2, “Elements of FCode Programming”, introduces the basic elements of FCode, stack notation, and programming style.
- Chapter 3, “Testing FCode Programs”, describes the process of producing FCode programs, from source file to testing working programs.
- Chapter 4, “Packages”, describes the basic units of FCode program function.
- Chapter 5, “Properties”, describes properties, which define how an FCode device driver program “sees” the hardware that it controls.
- Chapter 6, “FCode Basic Concepts”, discusses concepts that are common to most or all FCode drivers.
- Chapter 7, “Block and Byte Devices” through Chapter 11, “Memory-Mapped Buses” describe currently-defined device types, programming requirements, and give some examples of device drivers for the various device types.
- Chapter 12, “Open Firmware Dictionary”, describes currently-defined FCode words, their functions and use, with brief programming examples.
- Appendix A, “FCode Reference”, lists all currently-defined Fcode words according to functional grouping, name, and byte value.

- Appendix B, “Coding Style”, contains an Open Firmware coding guideline.

Related Books and Specifications

This manual does not pretend to cover everything you need to know to write FCode drivers for PCI cards. You’ll have to read some other books, too.

For information about Open Firmware, see the following manuals and Internet resources:

- *IEEE Standard 1275-1994 Standard for Boot (Initialization Configuration) Firmware, Core Requirements and Practices (IEEE Order Number SH17327. 800-678-4333. US$87.)*
- Open Firmware “binding” documents are available by anonymous FTP from ftp://playground.sun.com/pub/p1275/bindings. Among the bindings that may be of interest is:
    - *PCI Bus Binding to IEEE Standard 1275-1994 1.6* (or later).
- Open Firmware “recommended practice” documents are also available by anonymous FTP from ftp://playground.sun.com/pub/p1275/practice. Among the recommended practice documents that may be of interest are:
    - *16-color Text Extension* 1.2 (or later).
    - *8-bit Graphics Extension* 1.2 (or later).
- *Open Firmware Command Reference*, FirmWorks PN 000-0000-0000006-02. US$50 plus shipping, handling and applicable sales tax.

Since Open Firmware is a living technology that is constantly being enhanced by the Open Firmware Working Group, you may want to monitor their FTP site and/or Web page at http://playground.sun.com/pub/p1275 for changes and additions to Open Firmware documentation. Working Group meetings are open to all interested parties. See the Working Group’s Web page for details.

For information about PCI, see the following manual:

- *PCI Local Bus Specification 2.*1 (or later). Available from the PCI Special Interest Group, Box 14070, Portland, OR 97214, 800-433-5177, 503-234-6762 (fax) PCI_SIG@ccm.jf.intel.com. US$25 plus shipping.

For more information about Forth and Forth programming, see:

- *Programming Languages - Forth*, American National Standards Institute, Inc.
- *Forth: A Text and Reference*, Mahlon G. Kelly and Nicholas Spies. Prentice Hall, 1986.
- *Starting FORTH*, Leo Brody. FORTH, Inc., second edition, 1987.
- *Forth: The New Model*, Jack Woehr. M & T Books, 1992.

Information about FirmWorks publications can be obtained with an email request to info@firmworks.com, from ftp://ftp.firmworks.com/pub/open_firmware/literature or from FirmWorks World Wide Web page at http://www.firmworks.com.

Development Tools

FirmWorks has available PCI FCode Program developer tools that include an Open Firmware FCode tokenizer and a BIOS compressor. In many cases, the compressor makes it possible to include an FCode driver with an existing BIOS image in a card’s existing PCI Expansion ROM without increasing the size of the Expansion ROM.

If you don't have the PCI Developer's Kit and would like more information about it, contact FirmWorks at info@firmworks.com.

What Typographic Changes and Symbols Mean

The following table describes the typeface changes and symbols used in this book.

Table A Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `autoexec.bat` file. Use `dir` to list all files.
`AaBbCc123`	What you type, contrasted with on-screen computer output	`C:\>` **`dir`**
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `del` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options.
Code samples are included in boxes and may display the following:		
`C:\>`	MS-DOS prompt	`C:\>`
ok	FirmWorks Open Firmware command prompt	ok
0>	Apple Open Firmware command prompt	0>
`%`	UNIX C shell prompt	`system%`
`$`	UNIX Bourne and Korn shell prompt	`system$`
`#`	Superuser prompt, all shells	`system#`

This manual follows a number of typographic conventions:

- Keys are indicated by their name. For example:

  Press the Return key.

- When you see two key names separated by a dash, press and hold the first key down, then press the second key. For example:

  To enter Control-C, press and hold Control, then press C, then release both keys.

  Although the keyname (i.e. C in the preceding example) is shown in uppercase, the actual keystroke may be lowercase.

- When you see two key names separated by a space, press and release the first key and then press and release the second key. For example:

  To enter Escape B, press and release Escape, then press and release B.

  Although the keyname (i.e. B in the preceding example) is shown in uppercase, the actual keystroke may be lowercase.

- In a command line, square brackets indicate an optional entry and italics indicate an argument that you must replace with the appropriate text. For example:

  `cd [`*directory*`]`

- The Open Firmware system prompts and responses shown in this manual are based upon those of the FirmWorks implementation. Other Open Firmware implementations may format prompts and responses differently.
- When a table is too large to fit on a single page, the row separator is omitted from the bottom of the portion of the table that fits on the first page. This is to alert you to the fact that the table continues on the following page.

  Similarly, the title line of the portion of the table on the second and succeeding pages contain the notation *(Continued)* to alert you to the fact that the table continues from the preceding page.

  For example, the first portion of such a split table would look like:

Table 1 Diagnostic Test Commands

Command	Description
`probe-scsi`	Identify devices attached to a SCSI bus.
`probe-scsi-all` [*device-path*]	Perform `probe-scsi` on all SCSI buses installed in the system below the specified device tree node. (If *device-path* is absent, the root node is used.)

while the second portion of the same table would look like:

Table 1 Diagnostic Test Commands *(Continued)*

Command	Description
`test` *device-specifier*	Execute the specified device's self-test method. For example: `test floppy` - test the floppy drive, if installed `test /memory` - test number of megabytes specified in the `selftest-#megs` NVRAM parameter; or test all of memory if `diag-switch?` is `true` `test net` - test the network connection
`test-all` [*device-specifier*]	Test all devices (that have a built-in self-test method) below the specified device tree node. (If *device-specifier* is absent, the root node is used.)

Were this table contained within a single page, it would look like:

Table 1 Diagnostic Test Commands

Command	Description
`probe-scsi`	Identify devices attached to a SCSI bus.
`probe-scsi-all` [*device-path*]	Perform `probe-scsi` on all SCSI buses installed in the system below the specified device tree node. (If *device-path* is absent, the root node is used.)
`test` *device-specifier*	Execute the specified device's self-test method. For example: `test floppy` - test the floppy drive, if installed `test /memory` - test number of megabytes specified in the `selftest-#megs` NVRAM parameter; or test all of memory if `diag-switch?` is `true` `test net` - test the network connection
`test-all` [*device-specifier*]	Test all devices (that have a built-in self-test method) below the specified device tree node. (If *device-specifier* is absent, the root node is used.)

1

PCI Cards and FCode

The Purpose of FCode

Each PCI card identifies itself with a set of up to 64, 32-bit “configuration registers”. The purpose of these registers is to provide a standard set of descriptive information in a known place. The configuration registers contain data identifying the type of card, its manufacturer and various other characteristics of the card. In addition, a PCI card can have an “Expansion ROM” containing additional information such as a BIOS extension for the card or an FCode program.

A BIOS extension provides a driver for the card to be used when the card is installed in a system that uses an Intel x86 compatible processor.

An FCode program provides, at a minimum, additional descriptive information beyond that provided by the configuration registers and can provide a processor-independent boot-time driver for use in Open Firmware-based systems. An FCode program can also contain (or can help the operating system to locate) processor-specific and operating system-specific OS drivers.

Locating the FCode Program

The first 16 PCI configuration registers are collectively known as the *configuration space header*. Included within this header is the *Expansion ROM base address* register. If Bit 0 of this register is reset, the PCI card has no expansion ROM. If Bit 0 is set, the PCI card has one or more Expansion ROMs whose base address is specified by Bits 11 - 31 of the register. The ROM(s) can contain several different images to accommodate different machine and processor architectures.

As shown in Figure 1, each such ROM image has a header record and a PCI Data Structure that together describe the image. The header record is located at the start of the image and contains a pointer to the PCI Data Structure. The PCI Data Structure, in turn, contains a number of fields including:

- A *code type* field
  This field identifies the type of code contained within the image
- An *image length* field
  This field defines the length of the image in integral multiples of 512 bytes.

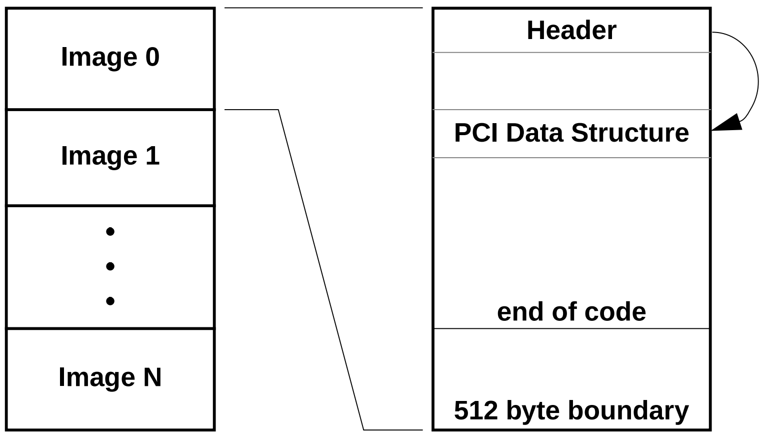

Figure 1 PCI Expansion ROM format

- An *indicator* field
  This field defines whether there are additional images located after this image.

FCode Program Functions

If the code type field has a value of 1, the ROM contains an FCode Program that, at a minimum, identifies the device and its characteristics.

An FCode Program may also include an optional software driver that lets you use the card as a boot device or a display device during booting. The software driver may also include diagnostic selftest code.

In addition to designing hardware, the process of developing PCI devices must include the writing, testing, and installing of an FCode driver for the device if it is to be used as a boot device in an Open Firmware-based system. These drivers, if present, serve three functions:

- To exercise the device during development, and to verify its functionality.
- To provide the necessary driver to be used by the system boot ROM during power-up.
- To provide device configuration information.

In practice, these functions overlap substantially. The same code needed by the system boot ROM usually serves to significantly test the device as well. The ROM code is used before and during the boot sequence. After the boot sequence finishes, and while not using the Open Firmware *User Interface*, most PCI devices are controlled with operating system device drivers.

Even if the PCI device is not a boot device, there are still advantages to providing a simple FCode driver that describes the characteristics of the device. Some operating systems (e.g. MacOS) are able to use this descriptive information to automatically attach operating system drivers to the device.

FCode Programs are written in the FCode programming language, which is similar to ANS Forth. FCode is described in more detail in Chapter 2 “Elements of FCode Programming”.

FCode ROM Format

An FCode ROM image is located within the PCI Expansion ROM on a 512 byte boundary. Its size typically ranges from 60 bytes (for a simple card that identifies itself but does not need a driver) to 1-4K bytes (for a card with a simple boot driver) to 10K bytes (for a device with a complex boot driver). It is good practice to make FCode boot drivers as short as is practical.

An FCode ROM image for PCI is organized as follows:

- Header (26 bytes: consisting of ROM signature and a pointer to the associated PCI Data Structure)
- PCI Data Structure (24 bytes: See the *PCI Local Bus Specification* for details)
- Body (FCode program; 0 or more bytes).
- End Token (either `end0`, a zero byte, or `end1`, an alternative all 1's byte).

Interpreting FCode

For each PCI slot containing a card, the following process is followed during boot-up to find and interpret any FCode programs:

- Scan all slots in numerical order.
- For each slot read the header type field.
  - If the header field type indicates a multi-function device, perform the following sequence for each function that is present.
  - Otherwise, perform the following sequence for the card's Function 0.
- Create a number of properties from the information contained in the PCI configuration registers. (See the *PCI Bus Binding to IEEE Standard 1275-1994* for the details.)
- Determine whether the device contains an expansion ROM and, if so, whether that ROM contains an image containing an FCode program.
- If an FCode program is present, copy the FCode program into RAM and evaluate it
- If the function does not have an FCode program:
  - Create the `"reg"` and `"name"` properties from the information in the PCI configuration registers.
  - If possible, create the `"power-consumption"` property from the state of the PRSNT1# and PRSNT2# connector.
- Disable fixed address response by clearing the PCI configuration address header's *command* register.
- Enable Memory Space response by setting Bit 1 in the command register.

Device Identification

An FCode ROM must identify its device. This identification must include, at a minimum, the device name. Identification information may include additional characteristics of the device for the benefit of the operating system and the CPU boot ROM.

The CPU's FCode interpreter stores each device's identification information in a *device tree* that has a node for each device. Each *device node* has a *property list* that identifies and describes the device. The property list is created as a result of interpreting the program in the FCode ROM.

Each property has a name and a value. The name is a string and the value is an array of bytes, which may encode strings, numbers, and various other data types.

See Chapter 5 “Properties” for more information.

Creating and Executing FCode Definitions

Many FCode programs create executable routines, called *methods*, that typically read from and write to device locations to control device functions. These definitions are also stored in the *device tree node* for that device.

Once defined, these routines may be executed under any of the following circumstances:

- Interactively through the Open Firmware User Interface’s `ok` prompt (for selftest or other purposes).
- By the Open Firmware *Client Interface* (for using this boot or display during system start-up).
- Automatically by the Open Firmware *Device Interface* during FCode interpretation (for power-on initialization or other purposes).

2

Elements of FCode Programming

FCode is a computer programming language defined by *IEEE Standard 1275-1994*. FCode is semantically similar to ANS Forth, but is encoded as a sequence of binary byte codes representing a defined set of Forth definitions.

FCode has these characteristics:

- The source format is machine and system independent.
- The binary format (FCode) is machine, system, and position independent.
- The binary format is compact.
- The binary format can be interpreted easily and efficiently.
- Programs are easy to develop and debug.
- The source format can easily be translated to binary format.
- The binary format can be translated back to source format.

Forth commands are called *words*, and are roughly analogous to procedures in other languages. Unlike other languages, such as C, which have operators, syntactic characters and procedures, in Forth every word is a procedure.

A Forth word is named by a sequence of between one to 31 printable characters. A Forth program is written as a sequence of Forth word names separated from one another by one or more “whitespace” characters (i.e. spaces, tabs or line terminators).

Forth uses a left-to-right reverse Polish notation, like some scientific calculators. The basic structure of Forth is: do this, now do that, now do something else, and so on.

New Forth words are defined as sequences of previously existing words. Subsequently, new words may be used to create still more words.

FCode is a byte-coded translation of a Forth program. Translating Forth source code to FCode involves replacing the Forth word names (stored as text strings) with their equivalent FCode numbers. The tokenized FCode takes up less space in ROM than the text form of the Forth program from which it was derived, and can be interpreted more easily and rapidly than the text form.

For purposes of this manual, the term FCode indicates both binary-coded FCode and the Forth programs written as ASCII text files for later conversion to binary-coded FCode.

Except where a distinction between the two forms is explicitly stated, the use of FCode in this manual can be assumed to apply equally to both FCode and Forth.

Colon Definitions

Three concepts are critical to understanding FCode (or Forth):

- A *colon definition* creates a new word with the same behavior of a sequence of existing words. A colon definition begins with a colon and ends with a semicolon.
- Once a new word has been created it is immediately available, either for direct execution or for use in future colon definitions.
- Most parameter passing is done through a pushdown, last-in, first-out *stack*.

Normally, the action associated with an FCode Function is performed when the FCode Function is encountered. This is called *interpret state*. However, you can switch from interpret state to *compile state*.

In interpret state, FCode Functions are executed as they are encountered. Interpret state operates until encountering a ":". The word ":" does the following:

- Allocates an FCode Number and associates it with the name immediately following the colon.
- Switches to compile state.

In compile state, FCodes are saved for later execution, rather than being executed immediately. The sequence thus compiled is installed in the action table as a new word, and can be later used in the same way as if it were a built-in word.

Compile state continues until a ";" is read. The word ";" does the following:

- Compiles an end-of-procedure FCode word
- Switches to interpret state

After compilation, the newly-assigned FCode word can be either interpreted or compiled as part of yet another new word.

If you define a new word having the same spelling as an existing word, the new definition supersedes the older one(s), but only for subsequent usages of that word.

Here's an example of a colon definition, defining a new FCode word `dac!`:

```
: dac!  ( data offset -- )  dac + rw! ;
```

Stack Operations

Each FCode word is specified by its effect on the stack and any side effects, such as accessing memory. Most FCode words affect only the stack, by removing arguments from the stack, performing some operation on them, and putting the result(s) back on the stack.

To aid understanding, conventional coding style requires that a *stack diagram* of the form `(  --  )` appear on the first line of every definition of a Forth word. The stack diagram specifies what the execution of the word does to the stack.

Entries to the left of `--` represent those stack items that the word removes from the stack and uses during its operation. The rightmost of these items is on top of the stack, with any preceding items beneath it. In other words, arguments are pushed onto the stack in left to right order, leaving the most recent one (the rightmost one in the diagram) on the top.

Entries to the right of `--` represent those stack items that the word leaves on the stack after it finishes execution. Again, the rightmost item is on top of the stack, with any preceding items beneath it.

In the previous example, the stack comment, beginning with "`(` " and ending with "`)`", shows that `dac!` takes two parameters from the stack, and doesn't replace them with anything when it's done.

You can place stack comments anywhere in a colon definition, and you should include them wherever they will enhance clarity.

Following the stack comment in the preceding example are a series of words that describe the behavior of `dac!`. Executing `dac!` is the same as executing the list of words in its colon definition.

Note that FCode words are separated by spaces, tabs, or newlines; "`( data`" is *not* the same as "`(data`". Any visible character is part of a word, and not a separator.

Although case is not significant, by convention FCode is written in lower case.

Data Types

The terms shown in Table 1 describe the data types used by Forth.

Table 1 Forth Data Type Definitions

Notation	Description
byte	An 8-bit value.
cell	The implementation-defined fixed size of a cell is specified in address units and the corresponding number of bits. Data-stack elements, return-stack elements, addresses, execution tokens, flags and integers are one cell wide. On Open Firmware systems, a cell consists of at least 32-bits, and is sufficiently large to contain a virtual address. The cell size may vary between implementations. A "32-bit" implementation has a cell size of 4. A "64-bit" implementation has a cell size of 8.
doublet	A 16-bit value.
octlet	A 64-bit value; only defined on 64-bit implementations,
quadlet	A 32-bit value.
double-cell number	A number represented by two cells. In memory, the cell at the lower address holds the more significant part of the number, and the address of that cell is used to identify the number. On the stack, the more significant part of the number is on top of the less significant part.

Additional Information

For more information about Forth programming needed to use available FCode primitives, refer to the Forth-related books listed in"Related Books and Specifications" on page xvi.

Programming Style

Some people have described Forth as a write-only language. While it sometimes ends up that way (like *any* badly written computer language), it *is* possible to write Forth (and FCode) programs that can be read and understood by more than just the original programmer. In fact, well-written Forth programs can be very clean and easy to understand. See Appendix B "Coding Style" for detailed information about the style used in the existing Open Firmware FCode source base.

Commenting Code

Comment code extravagantly, then consider adding more comments. The comments can help you and others maintain your code, and they don't add to the final size of the resulting FCode ROM.

Typical practice is to use "`(   )`" for stack comments and "`\`" for other descriptive text and comments.

In your comments, describe the purpose of your Forth words, their interface assumptions and requirements, and any unusual aspects of the algorithm you use. Try to avoid simply translating low-level details of your code into English. Comments like, "increment the variable" are rarely helpful.

Coding Style

Study the examples in this book to see an indentation and phrasing style that is widely used in Open Firmware source code. Adoption of that style will make your code more easily readable by the many FCode programmers who are accustomed to that style. If you are tempted to chose a different style, consider the following:

> Communication among humans is enhanced by adherence to a uniform set of conventions. No matter how "good" your custom style may be, it is unlikely that the existing body of Open Firmware source code and FCode Programs will be re-written in your new style.

Short Definitions

Keep word definitions short. If your definition exceeds half a page, try to break it up into two or more definitions. If it grows to a page or longer, you *should* break it up, if only to make the code easier to support in the future.

A *good* size for a word definition is one or two lines of code. Keeping definitions short and of limited functionality improves readability, speeds debugging and increases the likelihood that the word will be re-usable. Remember: re-use of Forth words is a principal contributor to compact ROM images.

Stack Comments

Always include stack comments in word definitions. It can be useful to compare intended function with what the code really does. Here's an example of a word definition with acceptable style.

```
\ xyz-map  establishes a virtual-to-physical mapping for each of the
\ useful addressable regions on the board

0 value status-register

: xyz-map  ( -- )

\ Base-address Size create-mapping then save virtual address

   my-address 4 map-low                           ( virtaddr )
   to status-register                             ( )
   my-address 10.0000 0 d+ frame-buf-size map-low ( virtaddr )
   to frame-buffer-adr                            ( )
;
```

Stack items are generally written using descriptive names to help clarify correct usage. See Table 2 for stack item abbreviations used in this manual.

Table 2 Stack Item Notation

Notation	Description
`|`	Alternate groups of stack results are separated by | surrounded with spaces. (in -- addr len false | result true) means either (in -- addr len false) or (in -- result true).
`|`	Individual stack item alternatives are separated by | without surrounding spaces. (in -- addr len|0 result) means either (in -- addr len result) or (in -- addr 0 result).
`???`	Unknown stack item(s).
…	Unknown stack item(s). If used on both sides of a stack comment, means the same stack items are present on both sides.
`< > <space>`	Space delimiter. Leading spaces are ignored.
`a-addr`	Variable-aligned address.
`addr`	Memory address (generally a virtual address).
`addr len`	Address and length for memory region.
`byte b`*xxx*	8-bit value (low order byte in a cell).
`char`	7-bit value (low order byte in a cell, high bit of low order byte unspecified).
`cnt len size`	Count or length.
`d`*xxx*	Double (extended-precision) numbers. 2 cells, high quadlet on top of stack.
`<eol>`	End-of-line delimiter.
`false`	0 (false flag).
`ihandle`	Pointer (handle) for an instance of a package.
`n n1 n2 n3`	Normal signed, one-cell values.
`nu nu1`	Signed or unsigned one-cell values.
`<nothing>`	Zero stack items.
`o o1 o2 oct1 oct2`	Octlet (64 bit signed value).
`oaddr`	Octlet (64-bit) aligned address.
`octlet`	An eight-byte quantity.

Table 2 Stack Item Notation *(Continued)*

Notation	Description
`phandle`	Pointer (handle) for a package.
`phys`	Physical address (actual hardware address).
`phys.lo` `phys.hi`	Lower/upper cell of physical address.
`pstr`	Packed string.
`quad` `q`*xxx*	Quadlet (32-bit value, low order four bytes in a cell).
`qaddr`	Quadlet (32-bit) aligned address.
`str`	Starting address of an unpacked string. Usually used in the form: xyz-str xyz-len
`{text}`	Optional text. Causes default behavior if omitted.
`"text<`*delim*`>"`	Input buffer text, parsed when command is executed. Text delimiter is enclosed in <>.
`[text<`*delim*`>]`	Text immediately following on the same line as the command, parsed immediately. Text delimiter is enclosed in <>.
`true`	-1 (true flag).
`u`*xxx*	Unsigned positive, one-cell values.
`ud`*xxx*	Unsigned positive double numbers. 2 cells, high quadlet on top of stack.
`virt`	Virtual address (address used by software).
`waddr`	Doublet (16-bit) aligned address.
`word` `w`*xxx*	Doublet (16-bit value, low order two bytes in a cell).
`x` `x1`	Arbitrary, one cell stack item.
`x.lo` `x.hi`	Low/high significant bits of a data item.
`xt`	Execution token.
`xxx?`	Flag. Name indicates usage (e.g. `done?` `ok?` `error?`).
`xy-str` `xy-len`	Address and length for unpacked string.
`xyz-sys`	Control-flow stack items, implementation-dependent.
`( C: -- )`	Control flow stack diagram. Used to describe the compile time behavior of words with different behaviors at compile-time and run-time.
`( -- )`	Run-time stack diagram.
`( E: -- )`	Execution stack diagram. Used with defining words to describe the run-time behavior of a word defined with that defining word.
`( R: -- )`	Return stack diagram.

A Minimum FCode Program

If a PCI card is not needed during the boot process, a minimal FCode program that merely declares the name of the device and the location and size of on-board registers will often suffice. Here is an example of an acceptable minimum program for a PCI card:

```
fcode-version2
" 0ABCDEF,bison" encode-string  " name" property

\ Create "reg" property

\ The first entry must be for the configuration space header
my-address my-space encode-phys
0 encode-int encode+ 0 encode-int encode+

\ The next N entries document the active base address registers
my-address my-space 0200.0010 or encode-phys encode+
0 encode-int encode+ 1000 encode-int encode+

\ The next entry describes the Expansion ROM base address register
my-address my-space 0200.0030 or encode-phys encode+
0 encode-int encode+ 8000 encode-int encode+

" reg" property
fcode-end
```

The following should be noted about the preceding example:

- `my-address` and `my-space` leave a total of three numbers on the stack representing the *phys.lo phys.mid phys.hi* address representation of a PCI node. The value of `"#address-cells"` is 3 for PCI which is reflected by this format.
- The *size* argument for `"reg"` is a double number encoded from two integers (e.g. 0 and 0 for the configuration space entry). This is due to the fact that the value of `"#size-cells"` is 2 for PCI which reflects PCI’s 64-bit address space.
- The configuration space entry in the `"reg"` property of a PCI device must have a size of zero.
- The second entry in the `"reg"` property assumes a device whose base address register at offset 0x10 in the configuration space header controls a 32-bit memory resource of size 4 KB. (See *PCI Bus Binding to IEEE Standard 1275-1994* for the encoding details.)
- The third entry in the `"reg"` property assumes a device whose PCI Expansion ROM is 32 KB in size.

A similar minimum program for an SBus device is:

```
fcode-version2
" 0ABCDEF,bison"  encode-string  " name" property

my-address h# 20.0000 + my-space h# 100 reg

fcode-end
```

The following should be noted about this SBus example:

- `my-address` and `my-space` each leave only a single number on the stack representing the *phys.lo phys.hi* address representation of an SBus node. The value of `"#address-cells"` is 2 for SBus which is reflected by this format.
- An offset of 0x200000 is being added to the value returned by `my-address`.
- The *size* argument of `"reg"` is a single number since `"#size-cells"` is 1 for SBus reflecting SBus's 32-bit address space.
- Since:
  - The value of `"#address-cells"` is 2 for SBus
  - The value of `"#size-cells"` is 1 for SBus
  - The format for an SBus `"reg"` (as defined in the binding *IEEE Draft Standard P1275.2/D14a Supplement for IEEE 1496 (SBus) Bus*) requires only a single entry

  the `reg` method can be used to create the `"reg"` property.

Note – The `reg` method is not useful in a PCI environment since `reg` can only work with buses whose `"#address-cells"` value is 2 and whose `"#size-cells"` value is 1. PCI is 3 and 2, respectively.
PCI also requires multiple entries within the `"reg"` property and `reg` can only encode a single entry.

If you wanted to add an address offset to a PCI device (as was done in the SBus example), you'd have to take care to add the offset to the *phys.lo* portion of the address while leaving *phys.mid* unaffected. This can be done most easily using double-precision addition as shown in the code fragment below:

```
\ The first entry must be for the configuration space header
my-address h# 20.0000 0 d+ my-space encode-phys
0 encode-int encode+ 0 encode-int encode+
```

Both of the example programs create a `"name"` property whose value is "`0ABCDEF,bison`" that will be used to identify the device. The `"name"` property's value should always begin with an identification of your company. The preferred form of this identification is the *organizationally unique identifier* (OUI), a sequence of six, uppercase hexadecimal digits which are assigned by the IEEE Registration Authority Committee. OUI's are guaranteed to be unique world-wide. (For more information about obtaining an OUI, please see the glossary entry for `"name"` in *IEEE Standard 1275-1994*.)

As an alternative to the OUI, you may use a sequence of from one to five uppercase letters representing the stock symbol of your company on any stock exchange whose symbols do not conflict with the symbols of the New York Stock Exchange and the NASDAQ Exchange. All stock exchanges in the United States satisfy this requirement. If a non-US company's stock is traded on US stock exchanges via "depository equivalents", those symbols also satisfy this requirement.

For those companies that have neither an OUI or a public stock symbol, an organizationally unique `"name"` property value must start with a company name that contains at least one lower case letter or that is longer than five characters thereby making it unlike a stock symbol (e.g. FirmWorks)

These sample programs could be extended by including additional code to declare additional properties, to create device methods, and/or to initialize the device after power-on.

FCode Classes

There are four general classes of FCode source words:

- **Primitives.** These words generally correspond directly to conventional Forth words, and implement functions such as addition, stack manipulation, and control structures.
- **System.** These are extension words implemented in the boot ROMs, and implement functions such as memory allocation and device property reporting.
- **Interface.** These are specific to particular types of devices, and implement functions such as `draw-character`.
- **Local.** These are private word definitions, implemented and used by devices.

Each FCode primitive is represented in the PCI card's ROM as a single byte. Other FCodes are represented in the PCI ROM as two consecutive bytes. The first byte, a value from 1 to 0x0f, may be thought of as an escape code.

One-byte FCode numbers range in value from `0x10` to `0xfe`. Two-byte FCode numbers begin with a byte in the range `0x01` to `0x0f`, and end with a byte in the range `0x00` to `0xff`. The single-byte values `0x00` and `0xff` signify "end of program" (either value will do; conventionally, 0x00 is used):

Currently-defined FCodes are listed in functional groups, in alphabetic order by name and in numeric order by FCode value in Appendix A, "FCode Reference".

Primitive FCode Functions

There are more than 300 primitive FCode functions, most of which exactly parallel ANS Forth words, divided into three groups:

- FCode words that generate a single FCode byte
- `tokenizer` macros
- `tokenizer` directives

Primitive FCode functions that have an exact parallel with standard ANS Forth words are given the same name as the equivalent ANS Forth word. Chapter 12 "Open Firmware Dictionary", contains further descriptions of primitive FCodes.

There are about another 70 `tokenizer` *macros*, most of which also have direct ANS Forth equivalents. These are convenient source code words translated by the tokenizer into short sequences of FCode primitives.

`tokenizer` *directives* are words that generate no FCodes, but are used to control the tokenization process. `tokenizer` directives include the words shown in Table 3.

Table 3 FCode Tokenizer Directives

Command	Stack Diagram	Description
`alias`	("new-name< >old-nmae< >" --)	Create a new command equivalent to an existing command.
`emit-byte`	(FCode# --)	Output specified FCode. Only works in tokenizer escape mode
`tokenizer[`	(--)	Enter tokenizer escape mode, allowing manual FCode generation.
`]tokenizer`	(--)	Exit tokenizer escape mode, resuming FCode interpretation
`external`	(--)	Words defined hereafter will be visible whenever this node is the current node.
`headerless`	(--)	Words defined hereafter will never be visible.
`headers`	(--)	Words defined hereafter will be optionally visible as a function of the setting of the configuration variable `fcode-debug?`.
`decimal`	(--)	When used outside of a colon definition, change the tokenizer's numeric conversion base to 10. When used inside a colon definition, append the phrase `d# 10 base !` to the word being defined.
`d#` *number*	(-- n)	Interpret *number* in decimal; base is unchanged.
`hex`	(--)	When used outside of a colon definition, change the tokenizer's numeric conversion base to 16. When used inside a colon definition, append the phrase `d# 16 base !` to the word being defined.
`h#` *number*	(-- n)	Interpret *number* in hex; base is unchanged.
`octal`	(--)	When used outside of a colon definition, change the tokenizer's numeric conversion base to 8. When used inside a colon definition, append the phrase `d# 8 base !` to the word being defined.
`o#` *number*	(-- n)	Interpret *number* in octal; base is unchanged.
`fload`	([filename<cr>] --)	Insert the specified file at this point.

System FCode Functions

System FCode functions are used by all classes of FCode drivers for various system-related functions. System FCode functions can be either *service* words or *configuration* words.

- Service words are available to the device's FCode driver when needed for functions such as memory mapping or diagnostic routines.
- Configuration words are included in the driver to document characteristics of the driver itself. These "properties" are made available for use by the operating system.

Interface FCode Functions

Interface FCode functions are standard routines used by the workstation's CPU to perform the functions of the PCI card's device. Different classes of devices will each use only the appropriate set of interface FCodes.

For example, in order to display a character on the screen, Open Firmware calls the interface FCode `draw-character`. Previously, the FCode driver for the device controlling that screen must have assigned a device-specific implementation to `draw-character`. It does this as follows:

```
: my-draw ( char -- ) \ "local" word to draw a character.
    …                 \ Definition contents.
;                     \ end of my-draw definition.
: my-install ( -- )   \ local word to install all interfaces.
    …
    ['] my-draw to draw-character
    …
;
```

When `my-install` executes, `draw-character` is assigned the behavior of `my-draw`.

Local FCode Functions

Local FCode functions are assigned to words defined within the body of an FCode Program. There are over 2000 FCode byte values allocated for local FCodes. The byte values are meaningful only within the context of a particular driver. Different drivers re-use the same set of byte values.

3 Testing FCode Programs

FCode Source

An FCode source file is essentially a Forth language source code file. The basic Forth words available to the programmer are listed in Chapter 12 “Open Firmware Dictionary”. Typically, Forth source files are named with a `.fth` suffix. FCode source files follow the same convention.

FCode programs have the following format:

```
\  Title comment describing the program that follows
fcode-version2
< body of the FCode program >
fcode-end
```

`fcode-version2` is a macro which directs the tokenizer to:

- Prepare the tokenizer to tokenize the following source text.
- Output the `start1` FCode.
- Output an FCode header. For a description of the FCode header see “FCode Binary Format” on page 18.

`fcode-end` is a macro that tells the tokenizer to:

- Output the `end0` FCode that marks the end of an FCode program. (`end0` *must* be at the end of the program or erroneous results may occur.)
- Stop tokenizing the current FCode program.
- Set the checksum and length fields of the FCode header to the program’s actual checksum and length.

The comment in the first line is not strictly necessary in many cases, but it is recommended since it allows some Open Firmware tools to recognize the file as a Forth source file. Enabling those tools to automatically recognize your file as a Forth source file may make your debugging easier.

Tokenizing FCode Source

The process of converting FCode source to FCode binary is referred to as *tokenizing*. A tokenizer program converts FCode source words to their corresponding byte-codes, as specified in Chapter 12 “Open Firmware Dictionary” and defined by *IEEE Standard 1275-1994*. A tokenizer program together with instructions describing its use is available from FirmWorks.

An FCode program’s source can reside in multiple files. The `fload` tokenizer directive directs the tokenizer input stream to load another file. `fload` acts like an `#include` statement in C. When `fload` is encountered, the tokenizer begins processing the file named by the `fload` directive. When the named file is completed, tokenizing continues with the file that issued the `fload`. `fload` directives may be nested.

Typically, the tokenizer produces a file in the following format:

- FCode header - 8 bytes
- FCode binary - remainder of file

The header format is system dependent.

You can use such a tokenized file to load either an FCode ROM or system memory for debugging as described in “Using the User Interface to Test FCode Programs” on page 26. Consult your tokenizer’s documentation for a description of how to produce ROMs from the file.

By convention, the file output by the tokenizer has the suffix `.fc`.

FCode Binary Format

The format of FCode binary that is required by the Open Firmware *FCode evaluator* is as follows:

Table 4 FCode Binary Format

Element	Structure
FCode header	Eight bytes
Body	0 or more bytes
End byte-code	1 byte, the `end0` byte-code

The format of the FCode header is:

Table 5 FCode Header Format

Byte(s)	Content
0	One of the FCodes: `version1` (not used with PCI), `start0`, `start1`, `start2`, `start4`
1	Format
2 and 3	16-bit checksum of the FCode body
4 through 7	Count of bytes in the FCode binary image including the header

The above information is presented for completeness. Since the tokenizer automatically generates this information, you will seldom be concerned about these details.

PCI Expansion ROM Header

As shown in Figure 1 on page 2, an FCode image that is stored in a PCI Expansion ROM must have some additional information included with it. This information can be synthesized by the tokenizer through the use of the `tokenizer[`, `emit-byte`, and `]tokenizer` directives as follows:

```
hex

tokenizer[
 55 emit-byte aa emit-byte \ PCI magic number
 34 emit-byte 00 emit-byte \ 0x16 Processor architecture unique data
 14 0 do 0 emit-byte loop  \ Pad bytes
 1c emit-byte 00 emit-byte \ Pointer to start of PCI Data Structure
 00 emit-byte 00 emit-byte \ Pad bytes
 \ Start of PCI Data Structure (DWORD aligned)
 ascii P emit-byte \ 4 Signature string “PCIR”
 ascii C emit-byte
 ascii I emit-byte
 ascii R emit-byte
 x1 emit-byte x2 emit-byte \ 2 Vendor ID = config reg 00/01
 y1 emit-byte y2 emit-byte \ 2 Device ID = config reg 02/03
 w1 emit-byte w2 emit-byte \ 2 Pointer to Vital Product Data
 18 emit-byte 00 emit-byte \ 2 PCI Data Structure length
 00 emit-byte \ 1 PCI Data Structure revision
 z1 emit-byte z2 emit-byte \ 3 Class Code = config reg 09/0a/0b
 z3 emit-byte
 q1 emit-byte q2 emit-byte \ 2 Image Length XXX - Must be fixed up
 r1 emit-byte r2 emit-byte \ 2 Revision Level of Code/Data
 01 emit-byte \ 1 Code Type 00=BIOS 01=OpenFW
 80 emit-byte \ 1 Indicator 00=another image 80=last image
 00 emit-byte 00 emit-byte \ 2 Reserved
]tokenizer

fcode-version2

fload mycode.fth

fcode-end
```

You will have to customize the “vendor ID”, “device ID”, “class code”, “image length”, “revision level” and “pointer to vital product data” fields appropriately for each FCode program.

Since this header is only required for your final ROM image, we suggest that you put your FCode source into one or more files (represented by the filename `mycode.fth` in the above example), and then `fload` the file(s) in a second file containing the PCI header code as shown above. You can then easily download `mycode.fth` with `dl` and can easily tokenize the same code by tokenizing the second file.

FirmWorks `pci-header` / `pci-header-end` Tokenizer Extensions

The FirmWorks tokenizer contains two additional directives named `pci-header` and `pci-header-end` that will create the PCI Expansion ROM header for you. The directive `pci-header` takes the “vendor ID”, “device ID” and “class code” field values from the stack and incorporates them into the header. `pci-header` puts a default value of 0 into the “pointer to vital product data” field, puts a default “1” in the “revision level” field and sets the “indicator” field to a default value of 1 indicating that this is the last image in the ROM. `pci-header-end` computes the length in bytes of the PCI Expansion ROM FCode image, divides that length by 512, rounds the result up and fills in the “image length” field. The following example creates such a header.

```
hex

\ pci-header ( vendor-id device-id class-code -- )
tokenizer[ v2v1 y2y1 z3z2z1 ]tokenizer pci-header

fcode-version2
fload mycode.fth
fcode-end
\ pci-header-end ( -- )
pci-header-end
```

Note – Calling `pci-header-end` without having first called `pci-header` will scramble the tokenizer’s output file since `pci-header-end` will “fix-up” a non-existent PCI header.

The FirmWorks tokenizer provides three additional directives for modifying the default values used by `pci-header`.

- `set-rev-level ( revision -- )`
- `set-vpd-offset ( addr -- )`
- `not-last-image ( -- )`

The following example shows how to set the “revision level”, “pointer to vital product data” and “indicator” fields to something other than their default values.

```
hex

tokenizer[ r2r1 ]tokenizer set-rev-level
tokenizer[ w2w1 ]tokenizer set-vpd-offset
not-last-image
tokenizer[ v2v1 y2y1 z3z2z1 ]tokenizer pci-header

fcode-version2
fload mycode.fth
fcode-end
pci-header-end
```

Testing FCode Programs on the Target Machine

Once you have created the FCode binary you can test it using the Open Firmware User Interface. The User Interface provides facilities to allow you to load your program into system memory and direct the FCode evaluator to interpret it from there. This allows you to debug your FCode without having to create a ROM and attach it to your plug-in board for each FCode revision during the debug process. See *IEEE Standard 1275-1994* for the complete specification of the User Interface.

The FCode testing process generally involves the following steps:

1. Configuring the target machine. This includes installing the hardware associated with the FCode program into the target machine and powering-up the machine to the User Interface.
2. Loading the FCode program into memory from a serial line, a network, a hard disk, or a floppy disk.
3. Interpreting the FCode program to create a *device node(s)* on the Open Firmware *device tree.*
4. Browsing the device node(s) to verify proper FCode interpretation.
5. Exercising the FCode program's device driver *methods* complied into the device node, if any.

If the FCode program does not include any methods which involve using the actual hardware (e.g. a driver which only publishes properties) then the program can be tested without installing the hardware.

Configuring the Target Machine

Setting Appropriate Configuration Parameters

Before powering-down the target machine to install the target hardware, a few NVRAM configuration variables should be set to appropriate values. You can set them from the User Interface as follows:

```
ok setenv auto-boot? false
ok setenv fcode-debug? true
```

Setting `auto-boot?` to `false` tells Open Firmware not to boot the OS upon a machine reset but rather to enter the User Interface.

The Open Firmware FCode evaluator always saves the names of any words created by interpreting FCode while the `external` tokenizer directive is in effect. By setting `fcode-debug?` to `true`, you tell the FCode evaluator also to save the names of words created while the `headers` tokenizer directive is in effect. (Words tokenized while the `headerless` directive is in effect are never saved.)

`fcode-debug?` defaults to `false` to conserve RAM space in normal machine operation. With names saved, the debugging methods described in later sections are easier to use since decompiled FCode will be displayed with names versus execution tokens.

“The Script” and the Open Firmware Startup Sequence

The configuration variable known as `nvramrc` is an area of NVRAM that is also known as the *“script”*. `nvramrc` can be used to store user-defined commands that are executed during start-up.

Typically, `nvramrc` is used by a device driver to save start-up configuration variables, to patch device driver code, or to define installation-specific device configuration and device aliases. It can also be used for bug patches or for user-installed extensions. Commands are stored in ASCII, just as the user would type them at the User Interface.

If the `use-nvramrc?` configuration variable is `true`, the script is evaluated during the Open Firmware start-up sequence as follows:

- Perform power-on self-test (POST) if present (system dependent)
- Perform system initialization
- Evaluate the script (if `use-nvramrc?` is true)
- Execute `probe-all` (i.e. evaluate FCode)
- Execute `install-console`
- Execute `banner`
- Execute secondary diagnostics
- Perform default boot (if `auto-boot?` is true)

Table 6 shows the top-level words that can be used in the script to control the overall execution of the start-up sequence.

Table 6 System Start-up Control Primitives

Command	Description
`probe-all`	Search for plug-in devices on the system-dependent set of expansion buses, creating device nodes for devices that are located. `probe-all` should not be executed more than once. It is normally executed during start-up following evaluation of the script, but this automatic execution is disabled if the script contains `banner` or `suppress-banner`.
`install-console`	Activate the console function and select input and output devices as follows: a. Activate the console so that subsequent input (e.g. `key`) and output (e.g. `emit`) will use the devices selected by `input` and `output`. b. Execute `output` with the value returned by `output-device`. c. Execute `input` with the value returned by `input-device`. d. If the preceding fails and if there is a *fallback* device for console functions, select that device as the console device. `install-console` may take other implementation-dependent actions to ensure the availability of a console in the event of failure and may display related messages.
`banner`	Display power-on banner. When included in the script, suppresses the execution of the sequence `probe-all install-console banner` in the system start-up sequence.
`suppress-banner`	When included in the script, suppresses the execution of the sequence `probe-all install-console banner` in the system start-up sequence.

Patching FCode of a Plug-in Card

It is sometimes desirable to modify the sequence `probe-all install-console banner`. For example, commands that modify the characteristics of plug-in display devices may need to be executed after the plug-in devices have been probed, but before the console device has been selected. Such commands would need to be executed between `probe-all` and `install-console`. Commands that display output on the console would need to be placed after `install-console` or `banner`.

Such custom control of the start-up sequence is accomplished by creating a script which contains either `banner` or `suppress-banner` since the sequence `probe-all install-console banner` is not executed if either `banner` or `suppress-banner` is executed from the script. This allows the use of `probe-all`, `install-console` and `banner` inside the script, possibly interspersed with other commands, without having those commands re-executed after the script finishes.

For example, assume that the plug-in device `/pci/framus` has an error in its `set-rate` method such that it divides by four when it should divide by two. This error could be patched using the script as follows:

```
ok nvedit
0: probe-all
1: dev /pci/framus
2: patch 2 4 set-rate
3: device-end
4: install-console banner
5: Control-C
ok nvstore
ok setenv use-nvramrc? true
ok reset-all
```

The script shown patches the broken plug-in card’s method by:

- First executing `probe-all` to cause all of the plug-in cards to be probed.
- Identifying `/pci/framus` as the device to be patched and patching its broken method.
- Ending the use of the `/pci/framus` device.
- Completing the startup sequence and suppressing the re-execution of the `probe-all install-console banner` sequence.

For more information about the script, the script editor and the high-level Forth patching facility, see *Open Firmware Command Reference* and/or “nvedit” on page 281. and “patch” on page 291.

Modifying the Expansion Bus Probe Sequence

The start-up sequence in the machine’s Open Firmware implementation normally examines all expansion buses for the presence of plug-in devices and their on-board FCode ROM program. It then invokes the FCode evaluator to interpret any such programs. This process is called *probing*.

When using the User Interface to load and interpret an FCode program in system memory, it is better to configure Open Firmware to avoid probing that device automatically. The probing can then be done manually (as explained later) from the User Interface.

Configuring an Open Firmware implementation to avoid probing a given slot on a given expansion bus can be done in various implementation-dependent ways. That is, they will be different for different systems and different expansion buses.

Many machines with a PCI bus have an NVRAM configuration variable named `pci-probe-list` that defines which PCI card slots will be probed during start up and the order in which they will be probed.

For example, a machine with four PCI slots might have the `pci-probe-list` configuration variable set to a default value of 0123. Setting `pci-probe-list` to 031 directs Open Firmware during start-up to probe first PCI slots 0, then slot 3, and finally slot 1. This re-arranges the order in which the slots are probed (perhaps to ensure that a particular graphics card is probed before all others) and leaves PCI slot 2 un-probed, free for use by the device under development.

Methods to prevent probing a given slot for other types of expansion buses can involve using the NVRAM `script`. An NVRAM script could patch an implementation-specific Open Firmware word that defines the bus's probe sequence or could modify a property of the expansion buses device node that describes the sequence.

After the FCode program is debugged and programmed in ROM on the device, you can do a full system test (including automatic probing of the new device), by restoring the expansion bus probing configuration to the default.

Getting to the User Interface

After completing the configuration described above, power-down the machine and install the device. Then power-up the system and it should stop at the ok prompt ready for User Interface commands.

Using the Command Line Editor of the User Interface

IEEE Standard 1275-1994 describes a required Command Line Editor and an optional Command Line Editor Extension to the User Interface. All implementations must support the following command line editing keys:

Table 7 Required Command Line Editor Keystroke Commands

Keystroke	Description
Delete	Erases previous character.
Backspace	Erases previous character.
Return (Enter)	Finishes editing of the line and submits it to the interpreter.

The standard also describes three groups of extensions of these capabilities (which are included in the FirmWorks implementation). The command line editing extension group includes the following command line editing keys.

Table 8 Optional Command Line Editor Keystroke Commands

Keystroke	Description
Control-B	Moves backward one character.
Escape B	Moves backward one word.
Control-F	Moves forward one character.
Escape F	Moves forward one word.
Control-A	Moves backward to beginning of line.
Control-E	Moves forward to end of line.

Table 8 Optional Command Line Editor Keystroke Commands

Keystroke	Description
Delete	Erases previous character.
Backspace	Erases previous character.
Control-H	Erases previous character.
Escape H	Erases from beginning of word to just before the cursor, storing erased characters in a save buffer.
Control-W	Erases from beginning of word to just before the cursor, storing erased characters in a save buffer.
Control-D	Erases next character.
Escape D	Erases from cursor to end of the word, storing erased characters in a save buffer.
Control-K	Erases from cursor to end of line, storing erased characters in a save buffer.
Control-U	Erases entire line, storing erased characters in a save buffer.
Control-R	Retypes the line.
Control-Q	Quotes next character (allows you to insert control characters).
Control-Y	Inserts the contents of the save buffer before the cursor.
Control-P	Selects and displays the previous line for subsequent editing.
Control-N	Selects and displays the next line for subsequent editing.
Control-L	Displays the entire contents of the editing buffer.

The command line history extension enables previously-typed commands to be saved in an emacs-like command history ring that contains at least 8 entries. Commands may be recalled by moving either forward or backward around the ring. Once recalled, a command may be edited and/or re-submitted (by typing the Return key). The command line history extension keys are:

Table 9 Optional Command Line History Keystroke Commands

Keystroke	Description
Control-P	Selects and displays the previous command in the command history ring.
Control-N	Selects and displays the next command in the command history ring.
Control-L	Displays the entire command history ring.

The command completion extension enables the system to complete long Forth word names by searching the dictionary for one or more matches based upon the already-typed portion of a word. After a user types in a portion of a word and types the command completion keystroke, the system behaves as follows:

- If the system finds exactly one matching word, the remainder of the word is automatically displayed.
- If the system finds several possible matches, the system displays all of the characters that are common to all of the possibilities.
- If the system cannot find a match for the already-typed characters, the system deletes characters from the right until there is at least one match for the remaining characters.
- The system beeps if it cannot determine an unambiguous match.

The command completion extension keys are:

Table 10 Optional Command Completion Keystroke Commands

Keystroke	**Description**
Control-Space	Complete the name of the current word.
Control-?	Display all possible matches for the current word.
Control-/	Display all possible matches for the current word.

Using the User Interface to Test FCode Programs

A synopsis of standard FCode words for downloading and executing FCode source files is shown Table 11. FirmWorks/Sun extensions are shown in Table 12

Table 11 File Download/Execute-related User Interface Commands

FCode	**Stack Notation**	**Function**
`begin-package`	*(arg-addr arg-len reg-addr reg-len path-addr path-len --)*	Creates a new node in the device tree in preparation for receiving FCode from the User Interface.
`end-package`	*(--)*	Completes a device tree entry and returns to the User Interface environment.
`open-dev`	*(path-addr path-len -- ihandle | 0)*	Opens the specified device node and all of its parents.
`close-dev`	*(ihandle --)*	Closes the specified device and all of its parents.
`device-end`	*(--)*	Unselect the active package, leaving none selected.
`set-args`	*(arg-addr arg-len reg-addr reg-len --)*	Sets values returned by `my-args`, `my-space` and `my-address` for the current node.
`execute-device-method`	*(… path-addr path-len cmd-addr cmd-len -- … ok?)*	Executes the named command within the specified device tree node.

Table 12 FirmWorks/Sun File Download/Execute-related User Interface Extensions

FCode	**Stack Notation**	**Function**
`select-dev`	*(path-addr path-len --)*	Opens the specified device node and all of its parents, and makes the device the current instance.
`select` *device-path*	*(--)*	Opens the specified device node and all of its parents, and makes the device the current instance.
`begin-select-dev`	*(path-addr path-len --)*	Opens all of the parents of the specified device node but does *not* call the device's `open` method.

Table 12 FirmWorks/Sun File Download/Execute-related User Interface Extensions

FCode	Stack Notation	Function
`begin-select`	*device-path (--)*	Opens all of the parents of the specified device node but does *not* call the device's `open` method.
`unselect-dev`	(--)	Closes the specified device node and all of its parents, and unselects the active package and current instance leaving none selected.

Using `dl` to Load From a Serial Port

`dl` can be used to load text files over a serial line connecting a "host" system (i.e. the system containing your source file) to a "target" system (i.e. the Open Firmware system on which you are going to test your code). `dl` loads text into memory until it receives a Control-D character (i.e. ASCII EOT, hex value 04), and then interprets the loaded text as Forth source code.

Many different communications programs will work with `dl`. The following example shows how to download a file using the Windows™ Terminal terminal emulator program on an MS-DOS® system.

1. Connect the target system's primary serial port to an available COM port, COM*n*, on the MS-DOS machine with a 3-wire "null modem" cable (i.e. a cable that connects Pin 3 to Pin 2, Pin 2 to Pin 3, and Pin 7 to Pin 7). For this example, we will assume the use of COM1 on the MS-DOS machine and TTYA on the target system.
2. Start Windows and open the Terminal application in the Accessories group. Check/correct the following menu settings (suggested values shown in italics):

   `Settings`
   - `Terminal Emulation`
     - *DEC VT-100 (ANSI)*
   - `Terminal Preferences`
     - `Terminal modes`
       - *Local echo off*
     - `CR -> CR/LF`
       - *Inbound off*
       - *Outbound off*
     - `Translations`
       - *None*
   - `Text Transfers`
     - *Standard Flow Control*
   - `Communications`
     - *Baud Rate: 9600*
     - *Data Bits: 8*
     - *Stop Bits: 1*
     - *Parity: None*
     - *Flow Control: None*
     - *Connector: COM1*

3. If you have not yet redirected the standard input and output to the target system's serial port, on the target's keyboard type:

```
ok ttya io
```

4. At the `ok` prompt in the Terminal window, type:

   `dl`

   which will produce the "Ready for download" prompt.

```
ok dl
Ready for download. Send file then type ^D
```

5. In the Terminal window, perform the following steps:
   a. Select "Send Text File" on the Transfers menu.
   b. Select the correct drive, directory and name of the file to be downloaded.
   c. In the "Following CR" section of the menu, turn *off* both "Append LF" and "Strip LF".
   d. Click "OK". This begins the transfer whose progress can be monitored in the status bar at the bottom of the Terminal program's window. The transfer is complete when the progress meter disappears. (If the file is short, the meter will come and go very quickly.)
6. Press Control-D in the Terminal window. After a delay while `dl` interprets the downloaded file, the `ok` prompt will appear in the Terminal window,

Downloading Multiple Files with `dl` and `fload`

Since `dl` merely downloads a text file without performing any interpretation of the text file until after the transfer is complete, any `fload` statements contained in a file downloaded with `dl` will not be processed correctly since the target system will not be able to locate the file associated with the `fload` statement.

A simple technique for solving this problem is to fragment your program into a series of files which, when concatenated in the proper order, construct your complete program file. None of these source files may contain any `fload` statements.

Once you have created such a series of files, the way to use them with `dl` depends upon the host's operating system.

MS-DOS®

1. Create a batch file that concatenates all of your files into a single file. For example:

```
rem This batch file concatenates the component files of the Phantom
rem driver.

echo hex > outfile.fth
echo 0 0 " 3" " /pci" begin-package >> outfile.fth
echo make-properties >> outfile.fth
copy outfile + header.fth + body.fth + trailer.fth outfile
echo assign-addresses >> outfile.fth
echo end-package >> outfile.fth
```

2. Use `dl` as described in the previous section.

Unix

1. Create a "load file" (in this example named `loadcoad.fth`) which consists of a series of `fload` statements that when executed sequentially will construct your complete program. For example:

```
\ id: loadcode.fth
\ purpose: Load file for the Phantom driver
\ copyright: Copyright (c) 1995 FirmWorks. All Rights Reserved.

hex

fload /home/code/header.fth
fload /home/code/body.fth
fload /home/code/trailer.fth
```

2. From this load file, create a "download file" (in this example named `dlcode`) to be used in the downloading process. To transform your first load file into the second form:
   - On any non-comment line of the file that is *not* an `fload` command, prefix a Unix `echo` command.

Note – Don't forget to surround your `begin-package` command with single quotation marks to prevent the shell from removing the double quotation marks in the arguments.

   - Insert a `begin-package` statement prior to the first `fload` command. This will automatically create a new device node into which to place your device methods.
   - Insert a `make-properties` statement immediately after the `begin-packages` command. This will automatically create the default PCI properties.
   - Insert an `assign-addresses` command after the last `fload` command. This will create the `assigned-addresses` property for your node, simulating the behavior of the PCI bus probing process in an Open Firmware system.
   - Insert an `end-package` statement after the `assign-addresses` command. This will end the creation of your new device node.

- Insert a Control-D after the `end-package` statement. This will cause `dl` to begin evaluating your FCode as soon as the downloading has been completed.

For example:

```
\ id: dlcode
\ purpose: Download file for the Phantom driver
\ copyright: Copyright (c) 1996 FirmWorks. All Rights Reserved.

echo hex

echo '0 0 " 3" " /pci" begin-package'
echo make-properties

fload /home/code/header.fth
fload /home/code/body.fth
fload /home/code/trailer.fth

echo assign-addresses
echo end-package
^D
```

Make the download file executable with `chmod` and `rehash`.

Note – See the next section for a detailed explanation of the use of `begin-package` and `end-package`.

---

3. Create a shell script named `fload` with the following contents:

```
#! /bin/sh
# This script enables the downloading of an FCode load file with dl
cat $1
```

Make this `fload` shell script executable with `chmod` and `rehash`, and store it in some directory on your search path. You'll be able to use this same shell script for all of your FCode development projects.

4. Assuming that you're using the `tip` terminal emulator program, type:

   `dl`

   which will produce the "Ready for download" prompt.

```
ok dl
Ready for download. Send file then type ^D
```

5. Type:

   ~C

which will produce a “local command” prompt from `tip`.

```
ok dl
Ready for download. Send file then type ^D
~C (local command)
```

Note – The `C` is case-sensitive and must be capitalized.

---

6. At the “local command” prompt, type:

   filename

   where *filename* is the name of the download file created in Step 2. Execution of *filename* sends all of your files over the serial link. When `dl` completes the download and evaluation, the `ok` prompt will be displayed.

```
ok dl
Ready for download. Send file then type ^D
~C (local command) dlcode
ok
```

A nice feature of this technique is that other versions of the load file can be easily created to accomplish other purposes. For example, to create a load file suitable for creating an FCode image, modify the load file as follows:

```
\ id: loadfc.fth
\ purpose: FCode load file for the Phantom driver
\ copyright: Copyright (c) 1995 FirmWorks. All Rights Reserved.

hex

fcode-version2

fload /home/code/header.fth
fload /home/code/body.fth
fload /home/code/trailer.fth

end0
```

Or to make an FCode image suitable for inclusion in a PCI expansion ROM, add the PCI Expansion ROM header as shown in “PCI Expansion ROM Header” on page 19.

Breaking your source files up into smaller files not only allows you to use these various downloading/tokenizing techniques, but it also makes it easier for you to re-use code if you dedicate each of your files to implementing methods that address a single problem category.

Using the User Interface to Interpret an FCode Program

FCode program interpretation involves creating a *device node* on the *device tree*. Device nodes are also known as *packages*. Creating a device node from downloaded FCode involves the following steps:

1. Setting up the environment with `begin-package`

For example, a `begin-package` call for creating a device node for a PCI card installed in PCI Slot 3 looks like:

```
ok 0 0 " 3" " /pci" begin-package
```

In the example, the string, `/pci`, indicates that the device node which will be created by the FCode program is to be a *child node* of the `/pci` node in the device tree.

In general, any device node that supports child nodes - called *parent* nodes - can be used as this argument to `begin-package`. The device node defined by the FCode program will be created as a child of that node. The full device pathname from the root node must be given.

In the example, the string `"  3"` indicates the PCI slot number, 3.

In general, this string is a pair of numbers separated by a comma. The first number identifies the PCI slot and the second number identifies the function number within that slot. The form of this physical address depends on the physical address space defined by the parent node. For children of a PCI node, the form is *slot-number, function-number.* Other types of parent nodes define different address spaces.

The physical address pair value is retrieved within the FCode program with both the `my-address` and `my-space` FCodes. The slot ID string is converted to a binary form consisting of three values. Those values can be retrieved with the FCode Program by using `my-address` for the *phys.lo* and *phys.mid* components and `my-space` for the *phys.hi* component. See *PCI Bus Binding to IEEE Standard 1275-1994* or Appendix C "PCI Bus Binding to Open Firmware" for a description of those three values.

In the preceding example, the initial `0 0` represents a null argument string passed to the FCode program. If you wish to pass a non-null argument string to `begin-package`, you must define that string in a Forth word and use the execution of that word to place the string on the stack. This is necessitated by the fact that Open Firmware systems are only required to provide two temporary buffers for the assembling of strings from the command line. With a non-null argument string, the `begin-package` command would require three temporary buffers. For example:

```
ok : testargs " framus" ;
ok testargs " 3" " /pci" begin-package
```

This argument string is retrieved within the FCode program with the `my-args` FCode. Generally, FCode programs do not take arguments at interpretation time so this will usually be the null string.

`begin-package` is defined as:

```
: begin-package  ( arg-str arg-len reg-str reg-len dev-str dev-len -- )
   select-dev new-device set-args
;
```

`select-dev` (parent-dev-str parent-dev-len --) - Opens the input device node (the parent node) and makes it the *current instance.* (See “Packages and Instances” on page 41 for a detailed description of *current instance*.)

`new-device` (--) - Initializes a new device node as a child of the currently active node and makes it the current instance.

`set-args` (arg-addr arg-len reg-addr reg-len --) - Sets the values returned by `my-args`, `my-space`, and `my-address` for the current instance.

2. Create default PCI properties with `make-properties`

`make-properties` simulates the behavior of the PCI bus probing process by creating default properties based upon the information found in the device’s PCI Configuration Space header.

This is a User Interface word intended by use in the context of `begin-package`... end-package *prior* to the evaluation of the FCode for the node.

3. Interpreting the loaded FCode with `byte-load`

`byte-load` is the User Interface command that invokes the FCode evaluator to compile the FCode program into the current instance.

For FCode programs downloaded with `load` or `dlfcode` use:

```
ok load-base ' c@ byte-load
```

`load-base` is the system default load address. The argument, `' c@` , tells `byte-load` to use `c@` as the access routine for reading the FCode.

4. Assign addresses to this node with `assign-addresses`

`assign-addresses` is used from the User Interface in the context of `begin-package` ... `end-package` to assign addresses to the current instance based on the current `"reg"` property value, and to create an `"assigned-addresses"` property reflecting those addresses. (The functionality of `assign-addresses` is normally automatically performed as part of the PCI probing process. However, this function must be done explicitly when a device node is being created manually with `begin-package` ... `end-package`.

5. Closing the environment with `end-package`

`end-package` closes the device tree entry started with `begin-package`.

```
ok end-package
```

Using the User Interface to Browse a Device Node

The User Interface has many built-in commands to navigate the device tree. Table 13 lists the User Interface commands supporting device node browsing:

Table 13 Commands for Browsing the Device Tree

Command	Description
`.properties`	Display the names and values of the active package's properties.
`dev` *device-path*	Read *device-path* from the input stream and make the specified device node the active package.
`dev` *node-name*	Search for a node with the given name in the sub-tree below the active package, and make the first such node found the active package.
`dev ..`	Make the parent of the active package the new active package.
`dev /`	Make the root machine node the active package.
`device-end`	Deactivate the active package, leaving no active package.
`"` *device-path*`" find-device`	Identical to `dev` except that *device-path* is specified as a string on the stack.
`get-inherited-property`	(*name-addr name-len -- true* | *value-addr value-len false*) Return property value of current instance or its parents
`get-my-property`	(*name-addr name-len -- true* | *value-addr value-len false*) Return property value of current instance.
`ls`	Display the names of the active package's children.
`pwd`	Display the device path name that names the active package.
`see` *wordname*	Decompile the specified word.
`show-devs [`*device-path*`]`	Display all the devices known to the system directly beneath a given level in the device hierarchy. `show-devs` used by itself shows the entire device tree.
`words`	Display the names of the active package's methods.

Once a device node has been created, you can use the User Interface to browse the node. See *IEEE Standard 1275-1994* for a more complete discussion on this. Below is a brief synopsis of the available commands.

- `show-devs` displays all known devices in the device tree.
- `dev` sets the active package to a named node so its contents can be viewed. For example, to make the ACME company's PCI device named "ACME,widget" the active package:

```
ok dev /pci/ACME,widget
```

- `find-device` is essentially identical to `dev` differing only in the way the input pathname is passed.

```
ok " /pci/ACME,widget" find-device
```

- `.properties` displays the names and values of all the properties created for the active package.
- `get-my-property` returns the value of the specified property from the active package.

- `get-inherited-property` returns the location and length of the property value array of the specified property from the active package or its parents. `dump` can then be used to display the property value array.
- `ls` displays the names of all child nodes, if any, of the active package.
- `see` *wordname* displays the source code (without comments) for *wordname*.
- `device-end` undoes the effects of the `dev` or `find-device` command by unselecting the previously-selected device and leaving no device selected.
- `pwd` displays the device pathname of the active package.
- `words` displays the names of the active package's words, if any. If there is no active package, `words` displays the names of all globally-visible words.

  The particular words displayed by `words` can be affected by the tokenizer directives `external`, `headers` and `headerless`, and by the state of the configuration variable `fcode-debug?`.

  If the FCode program was interpreted from text source, the tokenizer directives have no affect on the words that are displayed. However, if the FCode program is first tokenized and then evaluated, `words` displays:
  - All words which were created by the FCode evaluator while the tokenizer directive `external` was in effect.
  - All words created by the FCode evaluator while the tokenizer directive `headers` was in effect *if* the configuration variable `fcode-debug?` was true when the FCode was evaluated.

  `words` never displays words created by the FCode evaluator while the `headerless` tokenizer directive was in effect.

Using the User Interface to Test a Device Driver

The User Interface provides the capability to test the methods of an FCode program by allowing the user to execute individual methods from the User Interface prompt.

Device Node Methods

Using `select-dev`

`select-dev` initializes an execution environment for the methods of the package specified by its stack arguments. Although `select-dev` is not required by *IEEE Standard 1275-1994*, it can easily be synthesized if your implementation does not include it.

```
: select-dev  ( dev-str dev-len -- )
   2dup open-dev                     ( dev-str dev-len ihandle )
   dup 0= abort" Can't open device"  ( dev-str dev-len ihandle )
   to my-self                        ( dev-str dev-len )
   find-device
;
```

After executing `select-dev` you can execute the device node's methods directly by name. For example:

```
ok " /pci/ACME,widget" select-dev
```

`select-dev` performs the following:

- Effectively calls "`dev /pci/ACME,widget`" to make the named device the active package. This makes all the device methods "visible" to the User Interface.
- Establishes a chained set of package instances for each node in the path. In particular, this makes the package's instance-specific data items available to its methods.
- `select-dev` requires that each device node in the path, including the node under test, must have an `open` method.

Once these steps are performed you can execute the methods of the current device node by typing their name at the prompt. For example:

```
ok clear-widget-register
ok fetch-widget-register .
0
```

As is generally true of the Forth language, if execution of a method exposes an error in the code, the error can be isolated by executing the component words of the method step-by-step. Use `see` to decompile the method, and then type the component words individually until the error is evident. For example:

```
ok see clear-widget-register
: clear-widget-register
   enable-register-write
   0 widget-register rl!
   disable-register-write
;
ok enable-register-write
ok 0 widget-register rl!
ok disable-register-write
```

This process can be performed recursively by decompiling the component words and then individually executing their component words. This is much easier if most of the words were defined with the `headers` directive since `see` can then display the names of the component words instead of hexadecimal codes.

This process is also enhanced by executing `showstack`. `showstack` causes the stack's contents to be displayed prior to every ok prompt. For example:

```
ok 1 2
ok showstack
1 2 ok . clear 3 4
2
3 4 ok
```

If your Open Firmware implementation supports the Forth source level debugger, you can use it to step through your program and test it. (For more information on the debugger, see “The Forth Source-level Debugger” on page 120 of *Open Firmware Command Reference*.)

Device nodes can also be modified “on-the-fly” with any of the following techniques:

- Entering new methods definitions. These methods are compiled into the device node like the methods in the FCode program that created the node.
- Redefining a method to include some function neglected in the first definition. (Words that were previously defined using the original definition of the method are unaffected.) For example:

```
ok : open  open initialize-widget-register-2 ;
```

  In general, such redefinitions affect only external uses of the named method (i.e. calls from other packages via `$call-method` and the like) and interactive use via the User Interface. Previously compiled calls to the method within the same package are unaffected unless the method is called by name (e.g. with `$call-self`).

- Use `patch` to edit word definitions. Such patches affect all uses of the method, both internal and external. (See “patch” on page 291 for information on how to use this command.)

Resetting the machine causes all such corrections to be lost. Consequently, once your words are debugged you’ll probably want to include any modifications in the FCode program source.

`unselect-dev` reverses the effects of `select-dev` by calling the `close` method of each device in the path of the current active node, destroying the package instance of each node, and returning to the normal User Interface environment. Execute `unselect-dev` as follows:

```
ok unselect-dev
```

`unselect-dev` also is not required by *IEEE Standard 1275-1994*. Its definition is:

```
: unselect-dev  ( -- )  my-self close-dev  0 to my-self  device-end  ;
```

Using `select`

Some Open Firmware implementations provide the command `select` whose function is identical to `select-dev` except that `select` takes its argument from the command line rather than from the stack. For example:

```
ok select /pci/ACME,widget
```

Using `begin-select-dev`

Sometimes, `select-dev` will fail to work because the `open` method of a newly-written package does not work correctly. In such a case, `begin-select-dev` can be used since it does everything that `select-dev` does *except* for opening the last child node. For example:

```
ok " /pci/ACME,widget" begin-select-dev
```

`begin-select-dev` is not required by *IEEE Standard 1275-1994*. If your implementation does not include it, the same affect can be achieved with the following:

```
ok dev /<full-path-to-device>
ok : real-open  open ; \ Create an alias for the original open
ok : open  true ;     \ Create a dummy open that can’t fail.
ok " /<full-path-to-device>" select-dev
```

However, `begin-select-dev` has the advantage that, since it does not attempt to use the target’s `open` method, you don’t have to create a null `open` method which hides the very `open` method that you want to debug.

Using `begin-select`

Some Open Firmware implementations provide the command `begin-select` whose function is identical to `begin-select-dev` except that `begin-select` takes its argument from the command line rather than from the stack. For example:

```
ok begin-select /pci/ACME,widget
```

Using `execute-device-method`

`execute-device-method` executes a method directly from the normal User Interface environment. That is, it is not necessary to manually make the device node the current instance before executing the method. For example:

```
ok " /pci/ACME,widget" " test-it" execute-device-method
```

`execute-device-method` returns `false` if the method could not be executed; otherwise it returns `true` on top of whatever results were placed on the stack by the successful execution of the method.

`execute-device-method` performs the following steps:

- Establishes a chained set of package instances for each node in the path. In particular, this makes an instance of all data items of the device node available to its methods.
- Opens all device nodes in the name device path *except* the last device node in the pathname.
- Invokes the named method.

- Closes all the device nodes in the path (except the last one) destroying their package instances.
- Restores the current instance to the instance that was current prior to beginning this process.
- Restores the active package to the package that was active prior to beginning this process.
- Returns the results.

Note that, in contrast to `select-dev`, `execute-device-method` does not call the `open` method of the last device node in the path. Consequently, any method invoked in this manner must be able to stand alone i.e. not requiring any pre-established state which normally is created by `open` and not requiring `close` to be executed later to put the device back into a quiescent state.

In summary, `execute-device-method` is provided to allow execution of device node methods which have been designed to provide their own state initialization and therefore to execute without previous execution of the `open` method. A typical example is a `selftest` method (which may, in fact, call the `open` and `close` methods itself).

Using `apply`

`apply` provides an alternative manner of invoking `execute-device-method` in that `apply` takes its arguments from the input stream instead of from the stack. The above example would be invoked with `apply` as follows:

```
ok apply test-it /pci/ACME,widget
```

Since `apply` invokes `execute-device-method`, all of the restrictions listed above for `execute-device-method` must be followed.

Testing FCode Programs in Source Form

The User Interface enables you to skip the tokenizer and download FCode program source directly. This practice is very useful early in the development of an FCode program. However, there are some disadvantages:

- It may cause problems in the long run since generally the User Interface recognizes a larger number of words than does the FCode evaluator. So the FCode program developer who tests with FCode source may develop and test a program only to find that some of the words she used are not FCode words and will not be accepted by the tokenizer and the FCode evaluator. This will require the developer to rewrite code.
- To load source you should comment out `fcode-version2` and `end0`.
- Since the download commands accept only one file any `fload`ed files must be put in-line.

To load an ASCII Forth source file over a serial line, you use the command `dl`. In addition to loading the file over the serial line, `dl` compiles the Forth source while it is loading without requiring an extra command. See “Using dl to Load From a Serial Port” on page 27 for details on the use of `dl`.

Producing an FCode ROM

The output of the tokenizer program is used to make an actual FCode ROM. If your PROM burning tools do not accept the raw binary format of the tokenizer, you may need to develop a format conversion utility.

Exercising an Installed FCode ROM

You can either let Open Firmware automatically evaluate the FCode program from the ROM or you can remove the device from the Open Firmware probing as discussed earlier in “Configuring the Target Machine” on page 21.

The same process discussed for testing FCode programs that are loaded to system memory can be used to test FCode programs already loaded into ROM on the device.

If you take the device out of the probing sequence, a device node can be built manually as in the following example for a device installed in PCI slot 1:

```
ok 10000 constant rom-size
ok " /pci" select-dev
ok " 1" decode-unit          ( phys.lo phys.mid phys.hi )
ok rom-size  map-in          ( virt )
ok new-device                ( virt )
ok " " " 1,0" set-args       ( virt )
ok dup 1 byte-load           ( virt )
ok finish-device             ( virt )
ok rom-size   map-out
ok unselect-dev
```

This is essentially the same sequence as outlined for evaluating FCode loaded into system memory except that the user must map in and map out the FCode ROM by using the `decode-unit`, `map-in`, and `map-out` methods of the parent device node. For more information about these methods, see Chapter 11 “Memory-Mapped Buses”.

You can browse the device node and exercise the device methods in the same way as described earlier. You can also define new methods and patch existing ones. Of course these modifications will only remain until a system reset.

4 Packages

A *package* is the set of methods, data and properties that resides in a device node. In many cases, the terms “device node” and “package” can be used interchangeably; conventionally, “package” emphasizes the capabilities of and the interface presented by the set of methods, while “device node” emphasizes the device’s physical nature or its presence and location within the device tree.

Many packages implement a standard set of functions that provide a standard interface. Different packages often implement the same interface. For example, there might be two display device driver packages, each implementing the standard display device interface, but for two different display devices.

A *support package* is a group of functions, or methods, that implements a specific interface. A support package implements a library of functions that may then be called, as needed, by FCode programs and/or by other packages.

Support packages, which are provided by the system firmware, are independent of any particular hardware device, but are often related to a particular class of hardware device. For example, the `disk-label` support package provides services that are generally useful to device drivers for disk devices.

Packages and Instances

A package consists of three classes of information:

- *Methods*

  A set of software procedures that define the package’s behavior. Example: a disk driver would have a `read` method whose purpose is to read data from a disk into memory.

- *Properties*

  An externally visible list of names and associated values that identify the package and its associated device. Example: each package has a `name` property whose value is a text string giving the name of the package.

- *Data*

  Package data can be global (i.e. *static*) or private (i.e. *instance-specific*), and it can be initialized or zero-filled. The initial values of the initialized data are stored within the package definition. Example: A disk driver package might have an initialized, global variable used to keep track of whether the driver has been previously opened.

Packages exist regardless of whether or not the package is being used. The *active package* is the package whose methods and properties are currently visible (i.e. can be inspected from the User Interface). `dev` and `find-device` can be used to change the active package. However, being the active package only makes a package's methods visible; it does not enable the execution of those methods.

Before a package's methods may be executed, an *instance* of the package must be created. You can think of an instance as being a working copy of the package. Any number of package instances can be created from the same package.

By analogy, an instance is to a package as a multi-processing operating system process is to the file containing the process's executable image. In a multi-processing operating system, a new process is created each time a user executes a given program. Each such process contains the dynamically alterable data that is associated with the program. If the user decides to run the same program *N* times simultaneously, *N* processes will be running simultaneously, each with its own copy of the program's private data. When the user terminates a process, that process's copy of the data is destroyed and the memory used by that process is returned to the operating system without affecting any other copies of that same program that are running in separate processes.

Similarly, an instance is created from a package by "opening" that package. The act of opening a package allocates an *instance record* (i.e. a block of memory used to store the instance's data), sets the contents of the instance record to the initial, instance-specific values stored in the package, and returns an *ihandle* that is used to identify the instance subsequently. An instance record is shown in Figure 2

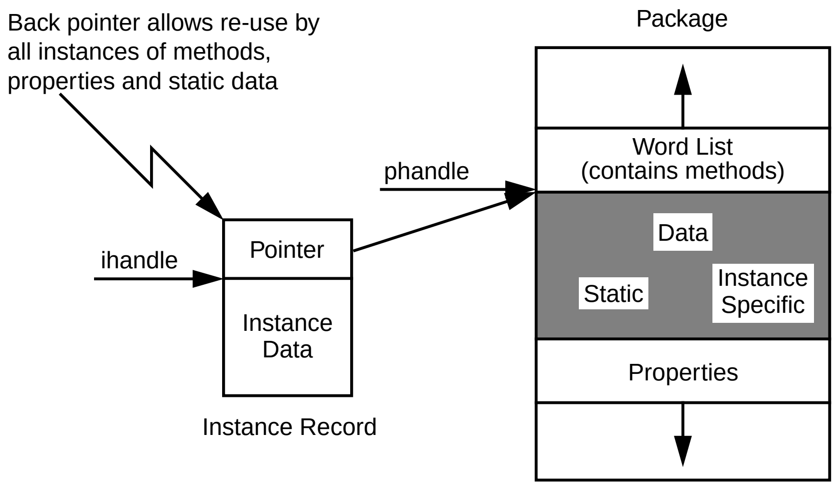
Back pointer allows re-use by all instances of methods, properties and static data
Package
Word List (contains methods)
phandle
Data
ihandle
Pointer
Static
Instance Specific
Instance Data
Properties
Instance Record

Figure 2 Relationship of Package to Instance Record

Just like the operating system process, multiple instances may be created from the same package, and may exist simultaneously. However, it is important to note that there is only one copy of static data and methods for a given package. All instances of the package use that common copy. Consequently, instances can pass information to each other through static data items. Instance-specific data, on the other hand, is private to each instance. Changes made to the instance-specific data in one instance record have no affect on any other instance.

Note – There is one exception to the last statement. A package for a plug-in device is created by evaluating the FCode of that device during the probing process. Since the evaluation process can execute device methods as well as define them, changes are sometimes made to the values of instance-specific data during FCode evaluation. At the end of the probing of the device, the instance-specific data values that exist at that time are stored in the package and are used as the initialization values for the instance-specific data for all subsequent usages of the package. An FCode driver can take advantage of this behavior to acquire information about a device's configuration in a particular system at probe time, and then pass this information to all subsequent instances.

An instance exists until it is terminated by "closing" it. When it is closed, the instance's instance record is freed and its ihandle becomes meaningless.

To use the methods or data of a package, an instance of that package must be (at least temporarily) the *current instance*. The current instance is the instance whose ihandle is stored in the value word `my-self`. The data and methods of the current instance may be called directly by name. The methods (and subsequently the data) of all other instances can only be called after identifying their instance with its ihandle (as with `$call-method`).

When a package method accesses a data item, it refers either to a static data item in the package or to the copy of an instance-specific data item that is stored in the current instance's instance record. The package method has access to the current instance's data; the data of all other instances is inaccessible.

A package to be opened is described by a device path or device alias. The process of opening the package includes opening each of the nodes in the device path from the root node to the specified device (i.e. from the top of the chain to the bottom). As each of these nodes is opened, an instance is created for the node and all of these instances are linked together in an *instance chain* as shown in Figure 3. When a method is accessed using the ihandle of the chain, each node in the chain is able to access the methods of its parent with `$call-parent` using the `my-parent` links provided by the instance chain. (See "Inter-package Calling Methods" on page 49.)

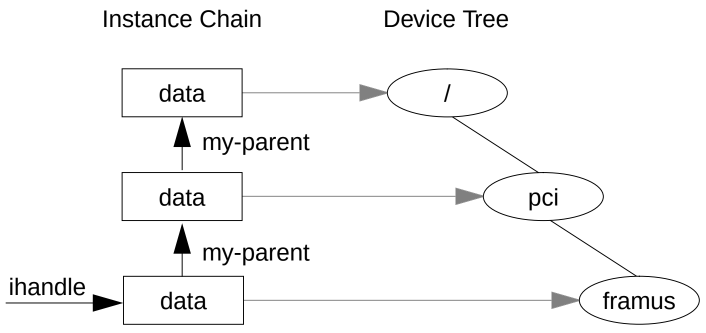

Figure 3 An Instance Chain for `/pci/framus`

When the chain is no longer needed, the individual instances of the chain may be closed by successive calling the `close` method of each node or the entire chain may be closed by calling `close-dev` with the ihandle of the chain. In either case, the chain is closed from bottom to top to enable a given node's `close` method to use parental methods.

The current instance is a very dynamic entity and is changed in several different ways under several different circumstances. Specifically:

- When a package is first created, `new-device`:
  - Creates a new device node that is a child of the currently active package.
  - Makes that new node the active package.
  - Makes that new node's instance the current instance.

  This causes any instance data / methods that are subsequently created (prior to the execution of `finish-device`) to be added to this node, and enables their later execution when an instance of this node is made current.
- When `open-dev` creates an instance chain, the current instance is repeatedly changed as each node of the instance chain is added to the instance chain (i.e. the root of the chain is first made current while it is being added to the instance chain, then the first child node is made current while it is added to the chain, and so on down to the leaf node). Immediately before terminating, `open-dev` restores the value in `my-self` to the value that `my-self` contained prior to the execution of `open-dev` and `open-dev` returns the ihandle of the leaf node of the newly-created instance chain. By manipulating the current instance in this way, `open-dev` is able to use instance-specific data as required.
- To execute a method not contained in the current instance, `$call-method` (or one of its derivatives) is used. `$call-method`:
  - Saves the current value of `my-self`.
  - Stores its *ihandle* argument in `my-self` (thus changing the current instance).
  - Executes the specified method.
  - Restores the saved value of `my-self`.
- From the User Interface, the current instance can be changed by the user by setting the value of `my-self` directly. This is most useful in a debugging scenario when testing the methods of an opened package. (The `select-dev` method discussed in "Using select-dev" on page 35 resets `my-self` for just this purpose.)

If a package is in the node `/packages`, `$open-package` can be used to create an instance of the package. Unlike packages opened with `open-dev`, packages opened with `$open-package` are opened by themselves without opening their ancestors. Each time a package instance is created by `$open-package`, that instance is attached to the instance that called `$open-package`. Figure 4 shows the modified instance chain that results when the `/pci/framus` instance opens the `obp-tftp` support package using `$open-package`.

Notice that the only additional instance that is created is one for the `obp-tftp` package, and that this instance is linked to the `/pci/framus` instance. If another instance of `obp-tftp` were opened by an instance in another instance chain, the resulting instance of `obp-tftp` would have no association with the instance shown in Figure 4.

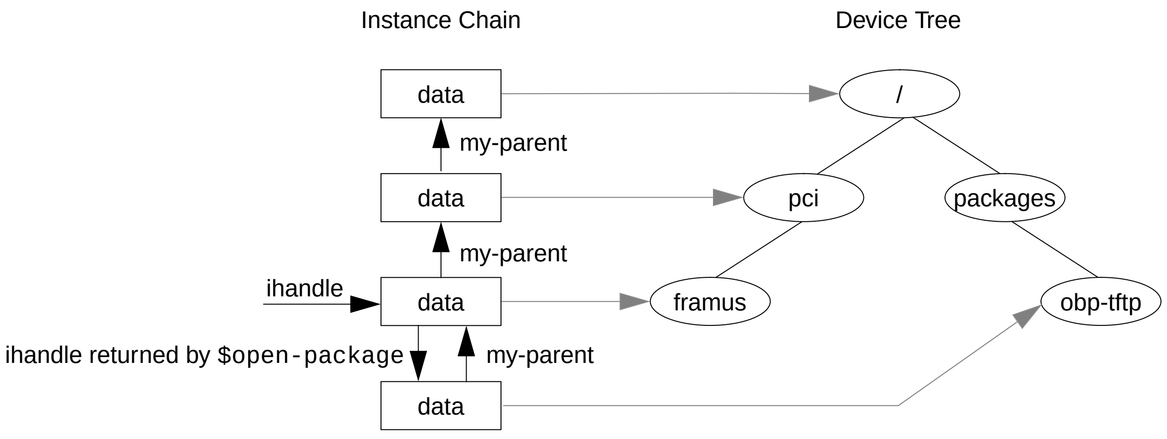

Figure 4 An Instance Chain for `/pci/framus` with `obp-tftp` Support

Package Data

Package data is named, read / write RAM storage used by package methods. Individual data items can be either "initialized" or "zero-filled" and either "static" or "instance-specific".

- "Static" data can be accessed at any time, regardless of whether or not the package has been opened. There is only one copy of each static data item, regardless of the number of currently-open instances of that package. The process of opening a package does not in itself alter the values of static data items (although you can, of course, write code to do so explicitly).
- "Instance-specific" data can only be accessed when a previously-opened instance of its package is the current instance. The process of opening a package creates copies of its instance-specific data items and establishes their initial values.
- "Zero-filled" data items are set to zero when a package is opened.
- "Initialized" data items are set to possibly-non-zero initial values when a package is opened. The initial values are established during the creation of the package.

Initialized data items are created by the Forth defining words `defer`, `value` and `variable`. Uninitialized data items are created by `buffer:`. Preceding the defining word with the Forth word `instance` causes the defining word to create an instance-specific item; otherwise the defining word creates a static data item.

Static data items are used for information that applies equally to all instances of the associated package. For example, virtual addresses of shared hardware resources, reference counts and hardware dependent configuration data are often stored as static data.

Instance-specific data items are used for information that differs between instances of the same package. For example, a package that provides a driver for a SCSI host adapter might have several simultaneous instances on behalf of several different target devices; each instance might need to maintain individual state information (e.g. the negotiated synchronous transfer rate) for its target.

Static and Instance-specific Methods

There are two kinds of package methods, depending on the environment in which they are called and their use of static and instance-specific data.

“Static methods” are methods that:

- Do not access instance-specific data either directly or by calling other instance-specific methods.
- Do not attempt to call methods of their parent.

Static methods can be called when there is no open instance of their package. When there is no instance, there is also no parent instance (which is the reason for the prohibition about calling parent methods).

The most important example of static methods is the `decode-unit` method which is called by the system during the process of searching the device tree without opening all of the nodes that are encountered.

“Instance-specific methods” are:

- Permitted to use instance-specific data
- Permitted to call the methods of their parent.

There is no structural difference between static and instance-specific methods. The concept of “static” methods is just a terse way of saying that some methods have to obey the restrictions outlined above. Instance-specific methods are the usual case; the static methods restrictions apply only to a very small set of special-purpose methods (typically residing in expansion bus device nodes).

Defining Methods, Properties and Data

It is possible to add new methods and new properties to a package definition at any time, even after the package definition has been completed. To do so, make the package the *active package* with `dev` or `find-device` and create the new definitions or properties. When you are finished, use `device-end` to unselect the active package leaving no package active. Generally speaking, the commands to do this would be put into `nvramrc`. (See ““The Script” and the Open Firmware Startup Sequence” on page 22.)

However, it is *not* possible to add new data items to a package definition after the package definition has been completed. This is due to the way in which package data is stored versus the way package methods and properties are stored.

Package methods and properties are stored in linked lists like a dictionary entry. They are linked into the method or property list of the package just like those methods and properties that were created with the original package.

Package data, on the other hand, is stored in a packed array associated with the package. This data storage method was chosen to improve the performance of data accesses.

At the time a package is first created, Open Firmware allocates a large, temporary data space that is used to hold data items during package definition. When the package definition is completed, Open Firmware collapses this temporary area to the minimum size necessary to hold the data items actually defined. Consequently, there is no place to store data items added later.

Note – If you attempt to define a new data item within a package, the Open Firmware implementation that you are using may *appear* to have created a new data item for you. However, you may also discover "incorrect" data behavior (e.g. data declared with `instance` behaves like static data). Attempting to add new data items to a package after the package has been defined will, at best, result in non-portable behavior.

Execution Tokens

A method is identified by its execution token, `xt`. For words in the package being defined, the Forth word `[']` returns an execution token. The execution token is returned by `find-method` for other packages. (See the following sections for more details.)

The execution token is used to execute a method in another package, and also to schedule a method for automatic, repeated execution by the system clock interrupt. See the `alarm` FCode.

Intra-package Calling Methods

A package can call its own methods directly simply by naming the target method in a Forth colon definition. Such calls require neither a call-time name search nor a change of the current instance. The binding of name to execution behavior occurs at compile time so subsequent redefinitions of a name do not affect previously-compiled references to old versions of that named method.

Infrequently, it may be desirable to call a method in the same package so that the name search happens at run-time. To do so, use either `$call-method` or `find-method`/`call-package` with `my-self` as the `ihandle` argument. (See the next section for details.)

Accessing Other Packages

Packages often use methods of other previously-defined packages. There are two types of packages whose methods can be used directly:

- The parent of the package being defined.
- Support packages in the `/packages` node of the device tree.

Phandles and Ihandles

A package definition is identified by its `phandle`. `find-package` returns the `phandle` of a package in the `/packages` node. The `phandle` can then used to open that support package or to examine its properties. For example:

```
" deblocker" find-package
```

returns either `false` (package not found), or `phandle true`.

Opening a support package with `open-package` returns an `ihandle`. This `ihandle` is used primarily to call the methods of the support package, and to close the support package when it is no longer needed.

An instance argument string must be supplied when opening any package (it may be null). The instance argument string can then be accessed from within the opened package with the `my-args` FCode (see below for details). For example (assume that phandle has already been found):

```
" 5,3,0" phandle open-package ( ihandle )
```

If the package cannot be opened, an `ihandle` of 0 is returned.

`$open-package` includes the functions of `find-package` and `open-package`. In most cases, it can be used in their place. The primitive functions `find-package` and `open-package` are rarely used directly, although `find-package` is sometimes used when it's necessary to examine a support package's properties without opening it.

The following FCode functions are used to find and open packages (within the `/packages` node):

Table 14 Package Access FCodes

Name	Stack Comment	Description
`find-package`	(name-str name-len -- false | phandle true)	Finds the package specified by the string *name-str name-len* within `/packages`. Returns the *phandle* of the package, or *false* if not found.
`open-package`	(arg-str arg-len phandle -- ihandle | false)	Opens an instance of the package *phandle*. Returns *ihandle* for the opened package, or *false* if unsuccessful. The package is opened with an instance argument string specified by *arg-str arg-len*.
`$open-package`	(arg-str arg-len name-addr name-len -- ihandle | false)	Shortcut word to find and open the package named *name-str name-len* within `/packages` in one operation. Returns *ihandle* for the opened package, or *false* if unsuccessful.

An example of using `$open-package` follows:

```
" 5,3,0" " deblocker" $open-package ( ihandle | 0 )
```

Table 15 Manipulating `phandles` and `ihandles`

Name	Stack Comment	Description
`my-self`	(-- ihandle)	Return the instance handle of the currently-executing package instance.
`my-parent`	(-- ihandle)	Return the instance handle of the parent of the currently-executing package instance.
`ihandle>phandle`	(ihandle -- phandle)	Convert an instance handle to a package handle.
`close-package`	(ihandle --)	Close an instance of a package.

Don’t confuse *phandle* with *ihandle*. Here’s how to use them:

1. Open the package with `$open-package` which returns an ihandle.
2. Use the ihandle to call the methods of the package.
3. When done calling the methods of the package, use the ihandle to close the instance of the package with `close-package`.

A package’s phandle is primarily used to access the package’s properties which are never instance-specific. Use `ihandle>phandle` to find the phandle of an open package. `my-self` and `my-parent` return ihandles, which can be converted into phandles with `ihandle>phandle`.

Inter-package Calling Methods

The following FCode functions enable the calling of methods of other packages:

Table 16 Method-Access Words

Name	Stack Comment	Description
`$call-method`	(… method-str method-len ihandle -- ???)	Shortcut word that finds and executes the method *method-str method-len* within the package instance *ihandle*.
`call-package`	(… xt ihandle -- ???)	Executes the method *xt* within the instance *ihandle*.
`$call-parent`	(… method-str method-len -- ???)	Executes the method *method-str method-len* within the parent’s package instance. Identical to calling `my-parent $call-method`.
`execute-device-method`	(… dev-str dev-len method-str method-len -- … false | ??? true)	Executes the method *method-str method-len* in the package named *dev-str dev-len*. Returns *false* if the method could not be executed.
`find-method`	(method-str method-len phandle -- false | xt true)	Finds the method named *method-str method-len* within the package *phandle*. Returns *false* if not found.

`$call-parent` is used most-often, but is the least flexible of the above methods; it is exactly equivalent to the sequence “`my-parent $call-method`”. Most inter-package method calling involves calling the methods of one’s parent; `$call-parent` conveniently encapsulates the process of doing so.

`$call-method` can call methods of non-parent packages. It is most commonly used for calling methods of support packages. The *ihandle* argument of `$call-method` identifies the package instance whose method is to be called.

For example:

```
$call-parent
$open-package $call-method
```

Both `$call-parent` and `$call-method` identify their target method by name. The *method-str method-len* arguments denote a text string that `$call-parent` or `$call-method` uses to search for a method of the same name in the target instance's list of methods. Obviously, this run-time name search is not as fast as directly executing a method whose address is already known. However:

a) Most packages have a relatively small number of methods,

b) Systems typically implement a reasonably-efficient name search mechanism, and

c) Inter-package calls tend to occur relatively infrequently.

Consequently, the length of time spent searching is usually not a limiting factor.

A more complete example demonstrates the use of `$open-package` and `$call-method`:

```
: add-offset  ( x.byte# -- x.byte#' )
    my-args " disk-label" $open-package  ( ihandle )
    " offset" rot                        ( name-addr name-len ihandle )
    $call-method
;
```

For those rare cases where method name search time is a limiting factor, use `find-method` to perform the name search once and then use `call-package` repetitively thereafter. `find-method` returns, and `call-package` expects, an "execution token" by which a method can be called quickly as shown in the following example that is somewhat faster if called repeatedly:

```
0 value label-ihandle \ Place to save the other package’s ihandle
0 value offset-method \ Place to save found method’s xt
: init ( -- )
    my-args " disk-label" $open-package ( ihandle ) to label-ihandle
    " offset" label-ihandle ihandle>phandle ( name-adr name-len phndle )
    find-method if
        ( xt ) to offset-method
    else
        ." Error: can’t find method"
    then
;
: add-offset ( d.byte# -- d.byte#’ )
    offset-method label-ihandle call-package
;
```

Because device access time often dominates I/O operations, the benefit of this extra code probably won't be noticed. It is only justified if the particular method will be called often.

Another use of `find-method` is to determine whether or not a package has a method with a particular name. This allows the addition of new methods to an existing package interface definition without requiring version numbers to denote which new or optional methods a particular package implements.

With `$call-method` and `$call-parent`, the method name search is performed on every call. Consequently, if a new method (either one with a new name or with the same name as a previously-existing name) is created, any subsequent uses of `$call-method` or `$call-parent` naming that method will find the new one. On the other hand, `find-method` "binds" a name to an execution token and subsequent redefinitions of that name do not affect the previous execution token, so subsequent uses of `$call-method` continue to call the previous definition. In practice, this difference is rarely important, since it is quite unusual for new methods to be created when a package is already open. The one case where methods are routinely redefined "on the fly" is when a programmer does it explicitly during a debugging session; doing such redefinitions is a powerful debugging technique.

All of the method calling functions described above change the current instance to the instance of the callee for the duration of the call, restoring it to the instance of the caller upon return.

`execute-device-method` and `apply`

In addition to the inter- and intra-package method calling techniques just described, there is another way of calling methods that is rarely used by FCode Programs. `execute-device-method` and its variant `apply` allow a user to invoke a method of a particular package as a "self-contained" operation without explicitly opening and closing the package as separate operations. `execute-device-method` first opens all the package's parents, then calls the named method, and then closes all the parents.

`apply` performs the same functions as `execute-device-method`, but it takes its arguments from the command line instead of from the Forth stack. It is consequently somewhat more convenient to use interactively.

`execute-device-method` and `apply` are most often used for methods like `selftest`. `selftest` methods are usually called with the `test` User Interface command which is usually implemented with `execute-device-method`.

Methods that are intended to be called with `execute-device-method` or its ilk must not assume that the package's `open` method has been called, because `execute-device-method` does not call the `open` method of the package containing the target method even though it opens all of the package's parents. Consequently, the target method must explicitly perform whatever initialization actions it requires, perhaps by calling the `open` method in the same package, or by executing some sub-sequence thereof. Before exiting, the target method must perform the corresponding `close` actions to undo its initialization actions.

`execute-device-method` was intentionally designed *not* to call the target's `open` and `close` methods automatically since the complete initialization sequence of `open` is not always appropriate for methods intended for use with `execute-device-method`.

In particular, an `open` method usually puts its device in a “fully operational” state, while methods like `selftest` often need to perform a partial initialization of selected device functions.

Although `execute-device-method` can be used with any “self-contained” operation, *IEEE Standard 1275-1994* specifies its use with the following methods:

- `selftest`
- `test`
- `test-all`

The FirmWorks implementation uses `execute-device-method` with the following additional methods:

- `abort?` (Used in the keyboard driver.)
- `clear` (Used in the keyboard driver.)
- `eject` (Used in the floppy driver.)
- `show-children` (Used by `probe-scsi` in the SCSI driver.)
- `watch-net` (Used in the Ethernet driver.)

Plug-in Device Drivers

Plug-in device drivers are plug-in packages implementing simple device drivers. The interfaces to these drivers are designed to provide basic I/O capability.

Plug-in drivers are used for such functions as booting the operating system from a device or displaying text on a device before the operating system has activated its own drivers. Plug-in drivers are added to the device tree during the probing phase of the Open Firmware ROM start-up sequence.

Plug-in drivers must be programmed to handle portability issues, such as hardware alignment restrictions and byte ordering of external devices. With care, you can write a driver so that it is portable to all of the systems in which the device could be used.

Plug-in drivers are usually stored in ROM located on the device itself, so that the act of installing the device automatically makes its plug-in driver available to the Open Firmware ROM.

For devices with no provision for such a plug-in driver ROM, the plug-in driver can be located elsewhere, perhaps in ROM located on a different device or in an otherwise unused portion of the main Open Firmware ROM. However, use of such a strategy limits such a device to certain systems and/or system configurations.

Common Package Methods

Different packages have different collections of methods depending upon the job(s) that the packages have to do. Having said that, the following four methods are found in many device drivers. None of them can be considered to be absolutely “required”, however, since the nature of a given driver governs the methods that the driver needs.

`open` and `close` are found in many drivers, but even they are not universally required. `open` and `close` are needed only if the device will be used with `open-dev` or another method that calls `open-dev`. Any device that has `read` and/or `write` methods needs `open` and `close`, as does any parent device whose children could possibly be opened.

Another way of looking at this is that `open` and `close` are needed for devices that are used to perform a series of related operations distributed over a period of time, relative so some other calling package. `open` initializes the device state that is maintained during the series of later operations, and `close` destroys that state after the series is complete.

To illustrate, a series of `write` calls generated by another package is such a series. Conversely, `selftest` is not such a series; `selftest` happens “atomically” as an indivisible self-contained operation.

Basic Methods

open (-- ok?)

Prepares a package for subsequent use. `open` typically allocates resources, maps, initializes devices, and performs a brief sanity check (making no check at all may be acceptable). `true` is returned if successful, `false` if not. When `open` is called, the parent instance chain has already been opened, so this method may call its parent’s methods.

close (--)

Restores a package to its “not in use” state. `close` typically turns off devices, unmaps, and de-allocates resources. `close` is executed before the package’s parent is closed, so the parent’s methods are available to `close`. It is an error to close a package which is not open.

Supplemental Methods

The following methods are highly recommended.

reset (--)

Puts a package into a quiescent state. `reset` is not invoked by any standard Open Firmware functions, but may be explicitly executed for “problem” devices in a particular Open Firmware implementation. `reset` is primarily for packages that do not automatically assume a quiet state after a hardware reset, such as devices that turn on with interrupt requests asserted.

selftest (-- error#)

Note – United States Patent No. 4,633,466, "Self Testing Data Processing System with Processor Independent Test Program", issued December 30, 1986 may apply to some or all elements of Open Firmware selftest. Anyone implementing Open Firmware should take such steps as may be necessary to avoid infringement of that patent and any other applicable intellectual property rights.

Tests the package. `selftest` is invoked by the Open Firmware `test` word. It returns 0 if no error found or a package-specific error number if a failure is detected.

`test` does not open the package before executing `selftest`, so `selftest` is responsible for establishing any state necessary to perform its function prior to starting the tests, and for releasing any resources allocated after completing the tests. There should be no user interaction with `selftest`, as the word may be called from a program with no user present.

If the device was already open when `selftest` is called, a new instance will still be created and destroyed. A well-written `selftest` should handle this possibility correctly, if appropriate.

If the device is already open, but it is not possible to perform a complete selftest without destroying the state of the device, the integrity of the open device should take precedence, and the selftest process should test only those aspects of the device that can be tested without destroying device state. The inability to fully test the device should not be reported as an error result; an error result should occur only if `selftest` actually finds a device fault.

The “device already open” case happens most commonly for display devices, which are often used as the console output device, and thus remain open for long periods of time. When testing a display device that is already open, it is not necessary to preserve text that may already be on the screen, but the device state should be preserved to the extent that further text output can occur and be visible after `selftest` exits. Any error messages that are displayed by the selftest method will be sent to the console output device, so when testing an already-open display device, such error messages should be avoided during times when `selftest` has the device in a state where it is unable to display text.

`selftest` is *not* executed within an `open/close` pair. Consequently, `selftest` should be written to do its own mapping and unmapping. When `selftest` executes, a new instance is created (and destroyed). It will have its own set of variables, values, and so forth. These quantities are not normally shared with an instance opened with the normal `open` routine for the package.

Package Data Definitions

The following examples show how to create static data items:

```
variable bar
5 value grinch
defer stub
create ival x , y , z ,
7 buffer: foo
ival foo 7 move           \ One way to initialize a buffer
```

The data areas defined above are shared among all open instances of the package. If a value is changed, for instance, the new value will persist until it is changed again, independent of the creation and destruction of package instances.

Any open instance of a package can access and change the value of a static data item, which changes it for all other instances.

The following examples show how to create instance-specific data items, whose values are not shared among open instances:

```
instance variable bar
5 instance value grinch
instance defer stub
7 instance buffer: foo
```

Instance-specific data areas are re-initialized every time a package instance is created (usually by opening the package), so each instance gets its own copy of the data area. For example, changes to *bar* in one instance will not affect the contents of *bar* in another instance. (Note that `create` operates across all the instances, and cannot be made instance-specific.)

The total amount of data space needed for a package's instance-specific data items is remembered as part of the package definition when `finish-device` finishes the package definition. Also, the contents of all the `variable`s, `value`s, and `defer`s at the time `finish-device` executes are stored as part of the package definition.

An instance of the package is created when that package is later opened. Data space is allocated for that instance (the amount of which was remembered in the package definition). The portion of that data space created with `variable`, `value`, or `defer` is initialized from the values stored in the package definition. Data space created with `buffer:` is set to zero.

You can add new methods and new properties to a package definition at any time, even after `finish-device` has been executed for that package. To do so, select the package and create definitions or properties.

However, it is *not* possible to add new data items to a package definition after `finish-device` has been executed for that package. `finish-device` sets the size of the data space for that package, and subsequently the size is fixed.

Note – If you attempt to define a new data item within a package, the Open Firmware implementation that you are using may appear to have created a new data item for you. However, you may also discover "incorrect" data behavior (e.g. data declared with instance behaves like static data). Attempting to add new data items to a package after the package has been defined will, at best, result in non-portable behavior.

Instance Arguments and Parameters

An instance argument (*my-args*) is a string that is passed to a package when it is opened. The string may contain parameters of any sort, based on the needs of the package, or may simply be a null-string if no parameters are needed. A null string can be generated with either `" "` or `0 0`.

The instance argument passed can be accessed from inside the package with the `my-args` FCode.

Note – A package is not required to inspect the passed arguments.

If the argument string contains several parameters separated by delimiter characters, you can pick off the pieces from within the package with `left-parse-string`. You can use any character as the delimiter; a comma is commonly used for this.

Note – Avoid using blanks or the `/` character, since these will confuse the parsing of pathnames.

A new value for `my-args` is passed every time a package is opened. This can happen under a number of circumstances:

1. The `my-args` string will be null when FCode on a PCI card is interpreted automatically by the Open Firmware system at power-on.
2. The `my-args` string is set by a parameter to `begin-package`, which is used to set up the device tree when Forth source code is downloaded and interpreted interactively.
3. The `my-args` string can be set with `set-args` before a particular slot is probed, if PCI probing is being controlled from `nvramrc`.

The above three instances happen only once, when the package FCode is interpreted for the first time. If you want to preserve the initial value for `my-args`, the FCode program should copy it into a static buffer to preserve the information.

Whenever a package is re-opened, a new value for `my-args` is supplied at that time. The method for supplying this new value depends on the method used to open the package, as described below.

1. The instance argument (`my-args`) is supplied as a string parameter to the commands `open-package` or `$open-package`.
2. User Interface commands, such as `open-dev`, `execute-device-method` and `test`, supply the entire pathname to the device being opened. This approach lets an instance argument be included within the pathname. For example, to open the PCI device “INTL,bwtwo” with the argument string “5,3,0”, enter:

```
ok " /pci/INTL,bwtwo:5,3,0" open-dev
```

A more complicated (and fictitious) example is the following:

```
ok " /pci/AAPL,fremly:test/grumpin@7,32:print/INTL,fht:1034,5"
ok open-dev
```

Here the string “test” is passed to the `AAPL,fremly` package as it is opened, the string “print” is passed to the `grumpin` package as it is opened, and the string “1034,5” is passed to the `INTL,fht` package as it is opened.

Package Addresses

Another piece of information available to a package is its address relative to its parent package. Again, there are two main ways to pass this address to the package:

- Part of the pathname of the package
- A string parameter given to the probe words

As an example of the first method, suppose the following package is being opened:

```
ok "/pci/scsi/disk@3,0:b" open-dev
```

Then the address of the `/disk` package relative to the `/scsi` package is `3,0`. Note that this address must match the initial value of the `"reg"` property (if present) of the `/disk` package.

The package can find its relative address with `my-unit`, which returns the address as a pair of numbers. The first number (*high*) is the number before the comma in the example above, and the second number (*low*) is the number after the comma. Note that these are numbers, not strings.

As an example of the second method, suppose a test version of an FCode package is being interpreted:

```
ok 0 0 " 3,0" " /pci" begin-package
```

Here the `my-args` parameters for the new FCode are null, the initial address is `3,0` and it will be placed under the `/pci` node.

The initial address can be obtained through `my-address` and `my-space`. Typically, you use `my-space` and `my-address` (plus an offset) to create the package's `"reg"` property, and also to map in needed regions of the device.

Package Mappings

Mappings set up by a package persist across instances unless they are explicitly unmapped. Passing the mapped addresses between instances is not usually worth the convolutions involved. It is usually better for each new instance to do its own mappings, being sure to unmap resources as they are no longer needed.

However, if it is unlikely that a particular package will have several open instances at the same time, it is usually a good idea to maintain only one mapping for all the open instances, using a reference counter to keep track of the number of open instances. The variables that store the reference counter and the mapped address must be static, not instance-specific. When the last instance is closed, the resources should be unmapped.

Modifying Package Properties

To modify the properties of a package, first make it the active package with `dev` or `find-device`. Then create or modify properties by executing `property` or one of its short-hand forms. When you are finished, use `device-end` to unselect the active package leaving no package active. Generally speaking, the commands to do this would be put into `nvramrc`.

See Chapter 5 "Properties“, for more information about properties.

Standard Support Packages

The `/packages` node of the device tree is special. It has children, but instead of describing a physical bus, `/packages` serves as a parent node for support packages. The children of `/packages` are general-purpose software packages not attached to any particular hardware device. The "physical address space" defined by `/packages` is a trivial one: there are no addresses. Its children are distinguished by name alone.

The children of `/packages` were created to simplify the job of writing FCode drivers for those device types that have a significant amount of work to do that is common to all devices of a given type and yet is not closely related to the hardware of any given device. By segregating this common code into the `/packages` node, individual FCode drivers are easier to write, are smaller in size, and are easier to debug since much of the work they must accomplish has been previously written and debugged.

For example, there is a significant amount of network protocol that must be implemented by every `network` device. Rather than make each network driver larger and more complex, the common functions were placed into the `obp-tftp` package for use by all `network` device drivers.

Like any other package, the children of `/packages` cannot be used until they are opened, and they must be closed when they are no longer needed. The FCodes `open-package`, `$open-package` and `close-package` are specifically provided for opening and closing the children of `/packages`; these FCodes work only with children of `/packages`.

IEEE Standard 1275-1994 defines three support packages that are children of `/packages`.

- `obp-tftp`
- `deblocker`
- `disk-label`

Each of these is described in the following sections.

TFTP Booting Support Package

`obp-tftp` implements the Internet Trivial File Transfer Protocol (TFTP). `obp-tftp` allows users to specify the use of "reverse address resolution protocol" (RARP) or the BOOTP protocol for use in address resolution. `obp-tftp` is typically used by a `network` device driver for its first stage network boot protocol.

`obp-tftp` implements three methods, `open`, `close` and `load` as shown in Table 17.

Table 17 TFTP Package Methods

Name	Stack diagram	Description
`open`	(-- okay?)	Prepares the package for subsequent use, returning *true* if the operation succeeds and *false* otherwise.
`close`	(--)	Frees all resources that were allocated by `open`.
`load`	(addr -- size)	Reads a client program from the default TFTP server, placing the program at memory address *addr* and returning its length *size*.

`open` and `close` are used as with any other package to prepare the package for use and to return the package to an unused condition when it is no longer needed. The `load` method, however, is the most interesting method defined by this package from the perspective of the FCode driver writer.

Instead of having to write the `load` method for a network device, the device's `load` method can be implemented simply by calling the `obp-tftp load` method using `$call-method` as shown in the following code fragment.

```
-1 instance value obp-tftp
: open  ( -- ok? )
   " obp-tftp" find-package if  ( phandle )
      my-args                   ( phandle arg$ )
      rot                       ( arg$ phandle )
      open-package              ( ihandle | 0 )
   else                         (   )
      0                         ( 0 )
   then                         ( ihandle | 0 )
   dup 0= if                    ( 0 )
      ." Can't open obp-tftp package" exit
   then                         ( ihandle )
   to obp-tftp                  (   )
   .
   .
   .
   true                         ( true )
;
: load  ( addr -- len )
   " load" obp-tftp $call-method  ( len )
;
```

To enable the use of the support package's `load` method, the driver must provide `read` and `write` methods for use by the support package's `load` method. For a more detailed explanation of the use of `obp-tftp`, see Chapter 8 "Network Devices".

Deblocker Support Package

The `deblocker` support package makes it easy to implement byte-oriented device methods, using the block-oriented or record-oriented methods defined by devices such as disks or tapes. It provides a layer of buffering between the high-level byte-oriented interface and the low-level block-oriented interface.

The `deblocker` support package implements the following methods:

Table 18 Deblocker Package Methods

Name	Stack diagram	Description
`open`	(-- okay?)	Prepares the package for subsequent use, allocating the buffers used by the deblocking process based upon the values returned by the parent instance's `max-transfer` and `block-size` methods. Returns *true* if the operation succeeds and *false* otherwise.
`close`	(--)	Frees all resources that were allocated by `open`.

Table 18 Deblocker Package Methods *(Continued)*

Name	Stack diagram	Description
`read`	(addr len -- actual)	Reads at most *len* bytes from the device into the memory buffer beginning at `addr`. Returns *actual*, the number of bytes actually read. If *actual* is zero or negative, the read operation failed. Uses the parent's `read-blocks` method as necessary to satisfy the request, buffering any unused bytes for the next request.
`write`	(addr len -- actual)	Writes at most `len` bytes from the device into the memory buffer beginning at `addr`. Returns *actual*, the number of bytes actually read. If *actual* is less than *len*, the `write` operation failed. Uses the parent's `write-blocks` method as necessary to satisfy the request, buffering any unused bytes for the next request.
`seek`	(pos.lo pos.hi -- status)	Sets the device position at which the next `read` or `write` will take place. Returns 0 or 1 if the operation succeeds and -1 if it fails.

`deblocker` (which is often used in combination with `disk-label`) is used to implement a block device's `read`, `write` and `seek` methods as shown in the following code fragment.

```
-1 instance value deblocker
: open  ( -- ok? )
   my-unit " set-address" $call-parent timed-spin if
      false exit
   then
   block-size to /block init-deblocker 0= if
      false exit
   then
   init-label-package 0= if
      deblocker close-package false exit
   then  true
;
: init-deblocker  ( -- ok? )
   " " " deblocker" $open-package dup to deblocker if
      true
   else
      ." Can’t open deblocker package" cr false
   then
;
: read  ( addr len -- #read )
   " read" deblocker $call-method
;
: write  ( addr len -- #written )
   " write" deblocker $call-method
;
: seek  ( pos.lo pos.hi -- status )
   offset-low offset-high d+  " seek" deblocker $call-method
;
```

To enable the `deblocker`, a device driver must provide the `block-size`, `dma-alloc`, `dma-free`, `max-transfer`, `read-blocks` and `write-blocks` methods. For a more detailed explanation of the use of `deblocker`, see Chapter 7 “Block and Byte Devices”.

Disk-Label Support Package

Disk (block) devices are random-access, block-oriented storage devices with fixed-length blocks. Disks may be subdivided into several logical “partitions”, as defined by a *disk label*—a special disk block, usually the first one, containing information about the disk. The disk driver is responsible for appropriately interpreting a disk label. The driver may use the standard support package `disk-label` if it does not implement a specialized label.

`disk-label` interprets the host system’s standard disk label, reading any “partitioning” information contained in it. It includes a first stage disk boot protocol for the standard label. In addition, in some systems (e.g. PowerPC systems) `disk-label` understands some set of file systems such that individual files can be accessed.

The `disk-label` support package implements the following methods:

Table 19 Disk Label Package Methods

Name	Stack diagram	Description
`open`	(-- okay?)	Prepare this package for subsequent use. Returns *true* if the operation succeeds and *false* otherwise.
`close`	(--)	Frees all resources that were allocated by `open`.
`load`	(addr -- size)	Reads a client program from the “standard” disk boot block location for the partition specified when the package was opened. Places the program at memory address *addr*, returning its length *size*.
`offset`	(d.rel-- d.abs)	Returns the 64-bit absolute byte offset *d.abs* corresponding to the 64-bit partition-relative byte offset *d.rel*. In other words, adds the byte location of the beginning of the selected partition to the number on the stack.

To enable `disk-label`, a device driver must provide the `read` and `seek` methods. Since `deblocker` is often used to implement those methods for a driver, `disk-label` and `deblocker` are often both used by a `block` device. For a more detailed explanation of the use of `disk-label`, see Chapter 7 “Block and Byte Devices”.

`disk-label` is used to implement a block device’s `load` and `offset` methods as shown in the following code fragment.

```
-1 instance value disk-label
: init-label-package
   0 to offset-high  0 to offset-low  my-args " disk-label" $open-package
   dup to disk-label if
      0 0 " offset" disk-label $call-method
      to offset-high  to offset-low  true
   else
      ." Can’t open disk label package" cr false
   then
;
: load  ( addr -- len )  " load" disk-label $call-method  ;
```

5 Properties

Properties describe characteristics of hardware devices, software and user choices. Properties are associated with the device node in which they are created and are accessible both by Open Firmware routines and by client programs. Properties can be inspected and, in some cases, modified.

Each property has a *property name* and a *property value*.

- *Property names* are human-readable strings consisting of one to 31 printable, lower-case letters and symbols not including "/", "\", ":", "[", "]" or "@". Property names beginning with "+" are reserved for future use by *IEEE Standard 1275-1994*
- *Property values* specify the contents, or value, of a particular property. The value is an array of bytes that may be used to encode integer numbers, text strings, or other forms of information.

Properties are accessed by name. Given a property's name, it is possible to determine whether that property has been defined and, if so, what its value is.

Property values are encoded as arrays of zero or more bytes for portability across machine architectures. The encoding and decoding procedures are defined by *IEEE Standard 1275-1994*. The encoding format is independent of hardware byte order and alignment characteristics. The encoded byte order is big-endian and the bytes are stored in successive memory locations without any padding.

The format of the property value array associated with a given property name is specific to that property name. There are five basic types of property value array formats:

- flag

  Since property value arrays may be of zero length, properties may convey "true" or "false" information by their presence or absence.

- byte

  An array of 1 or more bytes is stored in a property value array as a series of sequential bytes in the property value array.

- 32-bit integer

  A 32-bit integer is stored in a property value array in four successive bytes with the most significant byte of the integer in the next available address in the property value array followed by the high middle, low middle and least significant bytes of the integer (i.e. in big-endian format).

- text string

  A text string of n printable characters is stored in a property value array in n+1 successive locations by storing the string in the first n locations followed by a byte of zero value (i.e. a null terminated string).

- composite

  A composite value is made up of the concatenation of encoded bytes, encoded 32-bit integers and / or encoded strings. Each such primitive is stored immediately after the preceding primitive with no intervening space (i.e. the items are “packed”). Some examples of composite values are:

  - *physical address range*. Encoded as 4 integers: *phys.lo phys.mid phys.hi size*
  - *array*. The concatenation of *n* items of some type.
  - *structure*. The concatenation of an arbitrary collection of other types with no padding or internal alignment.

The standard defines a number of standard properties with specified names and value formats. If a package uses one of these standard properties then the value format of the property must be as defined by the standard. Packages may define other properties whose names do not conflict with the list of standard properties. Such newly defined properties may have any value format.

Properties may be created by FCode programs. The CPU’s Open Firmware understands certain property names that tell it such things as the type of a device (e.g. disk, tape, network, display, etc.). The firmware system uses this information to determine how to use the device (if at all) during the boot process.

Some operating systems understand other property names that give information used for configuring the operating system automatically. These properties include the driver name, the addresses and sizes of the device’s registers, and interrupt levels and interrupt vectors used by the device.

Other properties may be used by individual operating system device drivers. The names of such properties and the interpretation of their values is subject to agreement between the writers of the FCode programs and the operating system driver, but may otherwise be arbitrarily chosen. For example, a display device might declare width, height, and depth properties to allow a single operating system driver to automatically configure itself for one of several similar but different devices.

A package’s properties identify the characteristics of the package and its associated physical device, if any. You can create a property either with the `property` FCode, or with the `name`, `reg`, `model`, and `device-type` FCodes, described below.

For example, a framebuffer package might export its register addresses, interrupt levels, and framebuffer size. Every package has an associated property list, which is arbitrarily extensible. The user interface command `.properties` displays the names and values of the current node’s properties.

For example, if a property named foo is created in a device node which already has a property named foo, the new property supersedes the old one.

New properties can be added during the lifetime of a product. For backward compatibility, an FCode or device driver program that needs the value of a particular property should determine whether or not the property exists and, if not, the program should supply its own default value.

Standard FCode Properties

IEEE Standard 1275-1994 defines the following standard properties. A package should never create any property using any of the following names, unless the defined meanings and structures are used.

Standard Property Names

This group of properties applies to all device nodes regardless of type. The *"name"* property is required in all packages. The remaining properties are optional.

- `"name"`

  Defines the name of the package.

- `"reg"`

  Defines the package's address space(s).

- `"device_type"`

  Defines the characteristics that the device is expected to have.

- `"model"`

  Defines the manufacturer's model number.

- `"interrupts"`

  Defines the interrupts used by the device.

- `"address"`

  Specifies the virtual addresses of one or more memory-mapped regions of the device.

- `"compatible"`

  Specifies a list of devices with which this device is compatible.

- `"status"`

  Indicates the operational status of the device.

Display Device Properties

Display devices include bit-mapped frame buffers, graphics displays and character-mapped displays. Display devices are typically used for console output. The following properties are specific to display devices:

- `"big-endian-aperture"`

  Specifies the big endian aperture of the frame buffer.

- `"character-set"`

  Specifies the character set (e.g. ISO8859-1).
- `"depth"`

  Specifies the number of bits in each pixel of the display.
- `"height"`

  Specifies the number of pixels in the “y” direction of the display.
- `"linebytes"`

  Specifies the number of pixels between consecutive scan lines of the display.
- `"little-endian-aperture"`

  Specifies the little endian aperture of the frame buffer.
- `"width"`

  Specifies the number of pixels in the “x” direction of the display.

Network Device Properties

Network devices are packet-oriented devices capable of sending and receiving Ethernet packets. Network devices are typically used for booting.

- `"local-mac-address"`

  Specifies the pre-assigned network address.
- `"mac-address"`

  Specifies the last used network address.
- `"address-bits"`

  Specifies the number of address bits needed to address this device on the physical layer.
- `"max-frame-size"`

  Specifies the maximum packet size that the device can transmit at one time.

Memory Device Properties

Memory devices are traditional random-access memory, suitable for temporary storage of data.

- `"reg"`

  Specifies the physical addresses actually installed in the system.
- `"available"`

  Specifies the regions of physical addresses that are currently unallocated by Open Firmware.

MMU Properties

A memory management unit (MMU) is a device that performs address translation between a CPU’s virtual addresses and the physical addresses of some bus, typically the bus represented by the root node.

- `"available"`

  Specifies the regions of physical addresses that are currently unallocated by Open Firmware.

- `"existing"`

  Specifies all of the regions physical addresses actually installed in the system.

- `"page-size"`

  Specifies the number of bytes in the smallest mappable region of virtual address space.

- `"translations"`

  Describes the address translations currently in use by Open Firmware.

General Properties For Parent Nodes

- `"#address-cells"`

  Defines a device node's address format.

- `"#size-cells"`

  Specifies the number of cells that are used to encode the size field of a child's "reg" property.

- `"ranges"`

  Defines the relationship between the physical address spaces of the parent and child nodes.

Properties For PCI Parent Nodes

- `"#address-cells"`

  The value of this property for a PCI bus node is 3.

- `"#size-cells"`

  The value of this property for a PCI bus node is 2, reflecting the 64-bit address space of PCI.

- `"device_type"`

  The value of this property for a PCI bus node is “pci”.

- `"reg"`

  For nodes representing PCI-to-PCI bridges, the value denotes the Configuration Space address of the bridges's configuration registers. The format is the same as that for PCI child nodes.

  For nodes representing bridges from some other bus to PCI, the format is as defined for the other bus.

- `"bus-range"`

  Specifies the range of bus numbers controlled by this PCI bus.

- `"slot-names"`

  Describes the external labeling of add-in slots.

Properties for PCI Child Nodes

The following definitions are specified by the *PCI Bus Binding to IEEE Standard 1275-1994*.

- `"reg"`

  This standard property is mandatory for PCI Child nodes.

- `"interrupts"`

  The presence of this property indicates that the function represented by this node is connected to a PCI expansion connector’s interrupt line.

- `"alternate-reg"`

  Defines alternate access paths for addressable regions.

- `"has-fcode"`

  The presence of this property indicates that this node was created by the evaluation of an FCode program.

- `"assigned-addresses"`

  Defines the Configuration Space’s base address and size.

- `"power-consumption"`

  Describes the device’s maximum power consumption categorized by the various power rails and the device’s power-management state.

Each of the following PCI child node properties is created during the probing process, after the device node has been created, and before evaluating the device’s FCode (if any). The property values are those found in the standard PCI configuration registers.

Unless otherwise specified, each of the following properties has a property value created by encoding the value contained in the associated hardware register with `encode-int`.

- `"vendor-id"`
- `"device-id"`
- `"revision-id"`
- `"class-code"`
- `"interrupts"`

  This property is present only if the Interrupt Pin register is non-zero.

- `"min-grant"`
- `"max-latency"`
- `"devsel-speed"`
- `"fast-back-to-back"`

  This property is present only if the “fast-back-to-back” bit (Bit 7) of the function’s Status Register is set.

Detailed Descriptions of Standard Properties

"**#address-cells**"

This property applies only to bus nodes. It specifies the number of cells that are used to represent a physical address with a bus' address space. The value for PCI bus nodes is 3.

"**#size-cells**"

This property applies only to bus nodes. It specifies the number of cells used to represent the length of a physical address range (i.e. the "size" field of a child's `"reg"` property. The value for PCI bus nodes is 2.

"**address**"

This property declares currently-mapped device virtual addresses. It is generally used to declare large regions of existing mappings, in order to enable the operating system device driver to re-use those mappings, thus conserving system resources. This property should be created after virtual addresses have been assigned by mapping operations, and should be deleted when the corresponding virtual addresses are unmapped.

The property value is an arbitrary number of virtual addresses. The correspondence between declared addresses and the set of mappable regions of a particular device is device-dependent.

```
-1 value my-buffers
-1 value my-dma-addr
: map-me ( -- )
   my-address my-space 1.0000 " map-in" $call-parent  ( virt1 )
   to my-buffers
   2000 " dma-alloc" $call-parent  ( virt2 )  to my-dma-addr
   my-buffers encode-int  my-dma-addr encode-int  encode+
   " address" property
;
: unmap-me  ( -- )
   my-dma-addr 2000 " dma-free" $call-parent
   my-buffers 1.0000 " map-out" $call-parent
   " address" delete-property
;
```

See also: `free-virtual, property`

"**address-bits**"

This property, when declared in "`network`" devices, indicates the number of address bits needed to address this device on its network. Used as:

```
d# 48 encode-int " address-bits" property
```

See also: `property` and Chapter 8 "Network Devices".

"**alternate-reg**"

This property describes alternative access paths for the addressable regions described by the `"reg"` property. Typically, an alternative access path exists when a particular part of a device can be accessed either in memory space or in I/O space, with a separate base address register for each of the two access paths. The primary access paths are described by the `"reg"` property and the secondary access paths, if any, are described by the `"alternate-reg"` property.

If no alternative paths exist, the `"alternate-reg"` property should not be defined. If the device has alternative access paths, each entry (i.e. each *phys-addr size* pair) of its value represents the secondary access path for the addressable region whose primary access path is in the corresponding entry of the `"reg"` property value. If the number of `"alternate-reg"` entries exceeds the number of `"reg"` property entries, the additional entries denote addressable regions that are not represented by `"reg"` property entries, and are thus not intended to be used in normal operation; such regions might, for example, be used for diagnostic functions. If the number of `"alternate-reg"` entries is less than the number of `"reg"` entries, the regions described by the extra `"reg"` entries do not have alternative access paths. An `"alternate-reg"` entry whose *phys.hi* component is zero indicates that the corresponding region does not have an alternative access path; such an entry can be used as a “place holder” to preserve the positions of later entries relative to the corresponding `"reg"` entries. The first `"alternate-reg"` entry, corresponding to the `"reg"` entry describing the function’s Configuration Space registers, has a *phys.hi* component of zero.

The property value is an arbitrary number of (*phys-addr, size*) pairs where:

- *phys-addr* is (*phys.lo phys.mid phys.hi*), encoded with `encode-phys`.
- *size* is a pair of integers, each encoded with `encode-int`. The first integer denotes the most-significant 32 bits of the 64-bit region size and the second integer denotes the least-significant 32 bits thereof.

"**assigned-addresses**"

This property describes the location and size of regions of physical address space that are specified in the device’s Configuration Space base address registers.

The property value is zero to six (*phys-addr, size*) pairs where:

- *phys-addr* is (*phys.lo phys.mid phys.hi*), encoded with `encode-phys`.
- *size* is a pair of integers, each encoded with `encode-int`. The first integer denotes the most-significant 32 bits of the 64-bit region size and the second integer denotes the least-significant 32 bits thereof.

Each entry [i.e. (*phys-addr, size*) pair] in this property value corresponds to one (or two in the case of 64-bit-address Memory Space) of the function’s Configuration Space base address registers. The entry indicates the physical address that has been assigned to that base address register, and the size in bytes of the assigned region. The size is a power of two (since the structure of PCI Base Address registers forces the decoding granularity to powers of two). Please see the glossary entry for this property for a complete description of the formatting details.

Note: There is no implied correspondence between the order of entries in the `"reg"` property value and order of entries in the `"assigned-addresses"` property value. The correspondence between the `"reg"` entries and `"assigned-addresses"` entries is determined by matching the fields denoting the Base Address register.

"**available**"

This property defines the resources that are managed by this package (i.e. `/memory` or `/mmu`) that are currently available for use by a client program.

The property value is an arbitrary number of (*phys-addr*, *length*) pairs where:

- *phys-addr* is a *phys.lo phys.mid phys.hi* list of integers encoded with `encode-int`.
- *length* (whose format depends on the package) is one or more integers, each encoded with `encode-int`.

"**big-endian-aperture**"

This property is associated with `"display"` devices. Encoded identically to `"reg"` for the corresponding bus, the property value contains the address of the big endian “aperture” of the frame buffer (i.e. the address range through which the frame buffer can be addressed in big endian mode).

"**bus-range**"

This property specifies the range of bus numbers controlled by this PCI bus.

The property value is two integers, each encoded with `encode-int`. The first integer represents the bus number of the PCI bus implemented by the bus controller represented by this node. The second integer represents the largest bus number of any PCI bus in the portion of the PCI domain that is subordinate to this node.

"**character-set**"

This property, when declared in “`display`” devices, indicates the recognized character set for the device. The property value is a text string.

A typical value is “`ISO8859-1`”. 8859-1 is the number of the ISO specification for that particular character set, which essentially covers the full range of western European languages. To get a list of possible values, consult the X registry for which there is an address in the X11R5 documentation.

Used as:

```
" ISO8859-1" encode-string " character-set" property
```

See also: `property`, Chapter 10 “Display Devices”

"**class-code**"

This property contains the value of the “Class Code” register from the Configuration Space header. That register identifies the generic function of the device and (in some cases) a specific register-level programming interface.

The property value is the register’s value encoded with `encode-int`.

See also: *PCI Local Bus Specification*

"compatible"

This property specifies a list of devices with which this device is compatible. The property is typically used by client programs to determine the correct driver to use with the device in those cases where the client program does not have a driver which matches the "name" property.

The property value is the concatenation (with `encode+`) of an arbitrary number of text strings (encoded with `encode-string`) wherein each text string follows the formatting conventions as described for the `"name"` property.

Note – At the time of this writing, the Open Firmware Working Group is considering the adoption of a new "recommended practice" on the topic "Generic Names". Once this recommended practice is adopted, you are strongly encouraged to follow its recommendations which affect the usage of the `name` and `compatible` properties. Recommended practice documents can be obtained as described in "Related Books and Specifications" on page xvi.

See also: `"name"`

"depth"

This property is associated with `"display"` devices. Encoded with `encode-int`, the property value specifies the number of bits in each pixel of the display.

"device-id"

This property contains the value of the "Device ID" register from the Configuration Space header. That register identifies the particular device. The encoding of the register is determined by the device vendor.

The property value is the register's value encoded with `encode-int`.

See also: *PCI Local Bus Specification*

"device_type"

This property declares the type of this plug-in device. The type need not be declared, unless this device is intended to be usable for booting. If this property is declared, using one of the following key values, the FCode program *must* follow the required conventions for that particular type of device, by implementing a specified set of properties and procedures (methods). Used as:

```
" display" encode-string " device_type" property
```

Defined values for this property are:

Table 20 Standard Device Types

Device Type	Device Characteristics
`block`	Random-access, block-oriented device, such as a disk drive, usable as a boot file source. See Chapter 7 "Block and Byte Devices" for the requirements of this type of device.
`byte`	Random-access, byte-oriented device, such as a tape drive, usable as a boot file source. See Chapter 7 "Block and Byte Devices" for the requirements of this type of device.

Table 20 Standard Device Types *(Continued)*

Device Type	Device Characteristics
`display`	Framebuffer or other similar display device, usable for message display during booting. See Chapter 10 “Display Devices” for the requirements of this type of device.
`memory`	Random-access memory device. See *IEEE Standard 1275-1994* for the requirements of this type of device.
`network`	Packet-oriented network device, such as Ethernet, usable as a boot file source. See Chapter 8 “Network Devices” for the requirements of this type of device.
`pci`	A PCI bus node to which PCI plug-in devices can be attached. See Chapter 11 “Memory-Mapped Buses” for the requirements of this type of device.
`serial`	Byte-oriented device, such as a serial port, usable for console input and / or console output. See Chapter 9 “Serial Devices” for the requirements of this type of device.

See also: `device-type`, `property`

"**devsel-speed**"

This property contains the value of the “DEVSEL timing” field (Bits 9-10) of the “Status” register from the Configuration Space header. That field describes the timing of the DEVSEL# output of the device.

The property value is the register’s value encoded with `encode-int`. A value of 0 indicates “fast”, 1 indicates “medium” and 2 indicates “slow” timing.

See also: *PCI Local Bus Specification*

■ `"existing"`

Specifies all of the regions physical addresses actually installed in the system.

"**fast-back-to-back**"

This property should be present only if the “Fast Back-to-Back Capable” field (Bit 7) is set in the “Status” register from the Configuration Space header. That field indicates whether the device is capable of accepting fast back-to-back transactions when the transactions are not to the same agent.

See also: *PCI Local Bus Specification*

"**has-fcode**"

This property should be present only if the creation of this device node involved the evaluation of an FCode program as opposed to completely automatic creation from information in configuration registers.

"**height**"

This property is associated with `"display"` devices. Encoded with `encode-int`, the property value specifies the number of displayable pixels in the “y” direction of the display.

"**interrupts**"

For PCI devices, this property should be present only if the function represented by this node is connected to a PCI expansion connector’s interrupt line. The value of this property is determined from the contents of the “Interrupt Pin” register from the Configuration Space header.

The property value is the register's value encoded with `encode-int`. The defined values are:

Table 21 `"interrupts"` Property Value Encoding

Value	Description
1	The device uses the INTA# interrupt line
2	The device uses the INTB# interrupt line
3	The device uses the INTC# interrupt line
4	The device uses the INTD# interrupt line

The `"interrupts"` property is used to report the interrupt pin that the card uses, strictly within the domain of interrupts defined by the PCI specification.

It is the responsibility of the operating system's PCI bus driver code to translate the interrupts reported by its children into the interrupt domain of its parent.

This makes it possible to write portable, system-independent FCode drivers, because the FCode driver does not need to know system-specific information about the way that the system handles interrupts. The system-specific information is known by the code that handles the system component that actually performs the hardware mapping from PCI interrupt pins to whatever interrupt facitilies exist on the system.

In some cases, the mapping may even be hierarchical. For example, a NuBus-to-PCIBus bridge might translate PCI interrupt pins into NuBus interrupt vectors, then a VMEBus-to-NuBus bridge might translate NuBus interrupt vectors into VME levels, then a host-to-VMEBus bridge might translate VME levels into IRQs.

See also: *PCI Local Bus Specification*

"**linebytes**"

This property is associated with `"display"` devices. Encoded with `encode-int`, the property value specifies the number of pixels between consecutive scan lines of the display.

"**little-endian-aperture**"

This property is associated with `"display"` devices. Encoded identically to `"reg"` for the corresponding bus, the property value contains the address of the little endian "aperture" of the frame buffer (i.e. the address range through which the frame buffer can be addressed in little endian mode).

"**local-mac-address**"

This property, used with devices whose `"device_type"` is "`network`", should be present only if the device has a built-in, 48-bit, IEEE 802.3-style Media Access Control (MAC) address. The system may or may not use this address in order to access this device.

Used as:

```
" "(08,04,fe,23,46,9e)" encode-bytes " local-mac-address" property
```

See also: `mac-address`, `"mac-address"`, `property`, and Chapter 8 "Network

Devices".

"mac-address"

This property must be created by the `open` method of "`network`" devices to indicate the Media Access Control (MAC) address that this device is currently using. This value may or may not be the same as the `"local-mac-address"` property, if any. This property is typically used by client programs that need to determine which network address was used by the network interface from which the client program was loaded.

The property value is the six byte MAC address encoded with `encode-byte`.

Here's how it all fits together.

1. If a plug-in device has an assigned MAC address from the factory, this address is published as the value for `"local-mac-address"`.
2. The system (based on various factors such as presence or absence of `"local-mac-address"` and/or the value of the NVRAM parameter `"local-mac-address?"`) decides which address it prefers the plug-in device to use. The value returned by the `mac-address` FCode is set to this address.
3. The plug-in device then reports the address which it is actually using by publishing the `"mac-address"` property.

The following are code examples for three typical situations.

For a well-behaved plug-in "`network`" device (which has a factory-unique MAC address but can use another system-supplied MAC address if desired by the system), the FCode would appear as:

```
" "(08,04,fe,23,46,9e)"  encode-bytes  " local-mac-address"  property
mac-address              encode-bytes  " mac-address"        property
(plus code to "assign" the correct mac-address value into registers)
```

For a plug-in "network" device that has a factory-unique MAC address and is unable to alter its behavior to a different address, the FCode would appear as:

```
" "(08,04,fe,23,46,9e)" encode-bytes  " local-mac-address" property
" "(08,04,fe,23,46,9e)" encode-bytes  " mac-address"       property
```

For a plug-in "`network`" device which does not have any built-in MAC address, the FCode would appear as:

```
mac-address encode-bytes  " mac-address" property
(plus code to "assign" the correct mac-address value into registers)
```

See also: `mac-address`, `"local-mac-address"`, `property` and Chapter 8 "Network Devices".

"max-frame-size"

This property, when declared in "network" devices, indicates the maximum packet size (in bytes) that the physical layer of the device can transmit. This property is can be used by client programs to allocate buffers of the appropriate length.

Used as:

```
4000 encode-int  " max-frame-size" property
```

See also: property and Chapter 8 "Network Devices".

"max-latency"

This property contains the value of the "Max_Lat" register from the Configuration Space header. That register specifies how frequently the device needs to gain access to the PCI bus. The value is given in units of 250 nanoseconds. A value of zero indicates that the device has no major requirements for the setting of the Latency Timers.

The property value is the register's value encoded with encode-int.

See also: *PCI Local Bus Specification*

"min-grant"

This property contains the value of the "Min_Gnt" register from the Configuration Space header. That register specifies how long a burst period the device needs assuming a clock frequency of 33 MHz. The value is given in units of 250 nanoseconds. A value of zero indicates that the device has no major requirements for the setting of the Latency Timers.

The property value is the register's value encoded with encode-int.

See also: *PCI Local Bus Specification*

"model"

This property identifies the model name and number (including revision) for a device, for manufacturing and field-service purposes.

The "model" property is useful to identify the specific piece of hardware (the plug-in card), as opposed to the "name" property (since several different but functionally-equivalent cards would have the same "name" property, thus calling the same device driver). Although the "model" property is good to have in general, it generally does not have any other specific purpose.

The property value format is arbitrary, but conventional usage is to begin the string with the manufacturer's name (as with the "name" property) and to end the string with the revision level.

Used as:

```
" INTL,501-1415-1" encode-string " model" property
```

See also: "name", model, property

"**name**"

This property specifies the manufacturer's name and device name of the device. All device nodes *must* publish this property. The `"name"` property can be used to match a particular operating system device driver with the device.

The property value is an arbitrary string consisting of one to 31, case-sensitive letters, numbers and / or characters from the set { , . _ + - }. The string may contain at most one comma. Embedded spaces are not allowed.

IEEE Standard 1275-1994 specifies three different formats for the manufacturer's name portion of the property value.

For United States companies that have publicly listed stock, the most practical form of name is to use the company's stock symbol (e.g. INTL for Intel Corporation). This option is also available to any company anywhere in the world provided that there is no duplication of the company's stock symbol on either the New York Stock Exchange or the NASDAQ exchange. If a non-U.S. company's stock is traded as an American Depository Receipt (ADR), the ADR symbol satisfies this requirement. A prime advantage of this form of manufacturer's name is that such stock symbols are very human-readable.

An alternative is to obtain an *organizationally unique identifier* (OUI) from the IEEE Registration Authority Committee. This is a 24-bit number that is guaranteed to be unique world-wide. Companies that have obtained an OUI would use a sequence of hexadecimal digits of the form "0NNNNNN" for the manufacturer's name portion of the property. This form of name has the disadvantage that the manufacturer is not easily recognizable.

For those companies that neither have stock that trades publically on a U. S. stock exchange nor have an OUI, a name may be constructed that contains at least one lower case letter or is longer than five characters thereby making it unlike a stock symbol (e.g. Fujitsu).

Each manufacturer may devise its own format for the device name portion of the property value.

An example usage is:

```
" INTL,bison-printer" encode-string  " name" property
```

The `device-name` method may also be used to create this property.

Note – At the time of this writing, the Open Firmware Working Group is considering the adoption of a new "recommended practice" on the topic "Generic Names". Once this recommended practice is adopted, you are strongly encouraged to follow its recommendations which affect the usage of the `name` and `compatible` properties. Recommended practice documents can be obtained as described in "Related Books and Specifications" on page xvi.

See also: `device-name`, `property`, `compatible` "**page-size**"

This property specifies the number of bytes in the smallest mappable region of virtual address space managed by the `/mmu` package.

"**power-consumption**"

This property describes the device's maximum power consumption (in microwatts) categorized by the various power rails and the device's power-management state (standby or fully-on).

The property value is a list of up to ten integers encoded with `encode-int` in the following order:

- unspecified standby
- unspecified full-on
- +5V standby
- +5V full-on
- +3.3V standby
- +3.3V full-on
- I/O power standby
- I/O power full-on
- reserved standby
- reserved full-on

The "unspecified" entries indicate that it is unknown how the power is divided among the various rails. The "unspecified" entries must be zero if any of the other entries are non-zero. The "unspecified" entries are provided so that the `"power-consumption"` property can be created for devices without FCode, from the information on the PRSNT1# and PRSNT2# connector pins.

If the number of integers in the encoded property value is less than ten, the power consumption is zero for the cases corresponding to the missing entries. For example, if there are four integers, they correspond to the two "unspecified" and the two "+5" quantities, and the others are zero.

The following code would create a "power-consumption" property for a device with +5V standby consumption of 100 mA and +5V full-on consumption of 2.5A:

```
0 encode-int 0 encode-int encode+\ Set unspecified values to zero
500000 encode-int encode+        \ 100 mA@5V = 500,000 uW standby
12500000 encode-int encode+      \ 2.5A@5V = 12,500,000 uW full-on
" power-consumption" property
```

"**ranges**"

The `"ranges"` property is a list of child-to-parent bus-specific address translations required for most bus node devices.

`"ranges"` is a property for those bus devices whose children can be accessed with CPU load and store operations (as opposed to buses like SCSI, whose children are accessed with a command protocol).

The "ranges" property value describes the bus-specific address translation that defines the correspondence between the part of the physical address space of the bus node's parent available for use by the bus node (the parent address space), and the physical address space defined by the bus node for its children (the child address space).

The "ranges" property value is a sequence of (*child-phys*, *parent-phys*, *size*) specifications where:

- *child-phys* is a starting address in the child physical address space defined by the bus node.
- *parent-phys* is a starting address in the physical address space of the parent of the bus node.
- *size* is the length in bytes of the child's address range.

The specification means that there is a one-to-one correspondence between the child addresses and the parent addresses within that range. The parent addresses given are always relative to the parent's address space.

child-phys is an address in the child address space encoded with `encode-phys`. For PCI, this means an address specification of the form *phys.hi phys.mid phys.lo*.

parent-phys is an address in the parent address space encoded with `encode-phys`.

The number of integers in each *size* entry is determined by the value of the `#size-cells` property of the node in which the `ranges` property appears. In the case of PCI, *size* is a list of two integers. The integers of the *size* entry are encoded with `encode-int`.

For a PCI node in a PowerPC Reference Platform (PPCRP) compliant machine, the total size of each such specification is six 32-bit numbers (one for the parent address space, three for the child address space, and two for the size). Successive specifications are encoded sequentially. A space with length 2**(number of bits in a machine word) is represented with a size of 0.

It is recommended (and not required) that the specifications be sorted in ascending order of *child-phys*. The address ranges thus described need not be contiguous in either the child space or the parent space. Also, the entire child space must be described in terms of parent addresses, but not all of the parent address space available to the bus device need be used for child addresses (the bus device might reserve some addresses for its own purposes, for instance).

In the PPCRP machine example, consider a 4-slot 32-bit PCI bus attached to a machine whose physical address space consists of a 32-bit "memory" space (Bit 31 = 0) and a 32-bit "I/O" space (Bit 31 = 1).

- ISA I/O space appears in the parent's "I/O" space at 0x8000.0000 and has a size of 0x1.0000.
- A reserved block of addresses begins at 0x8001.0000 and has a size of 0x7f.0000.
- PCI configuration space begins at 0x8080.0000 and has a size of 0x80.0000. The configuration registers of the individual PCI slots appear at addresses 0x8080.1000, 0x8080.2000, 0x8080.4000, and 0x8080.8000.
- PCI I/O space begins at 0x8100.0000 and has a size of 3e80.0000.
- Parity/interrupt vectors begin at 0xbf80.0000 and have a size of 0x80.0000.
- PCI memory space begins at 0xc000.0000 and has a size of 3f00.0000.

The PCI device defines:

- Configuration spaces for Devices 1 through 4 that each begin at 0x0000.0000 and have a size of 0x100 bytes.
- ISA I/O space that begins at 0x0000.0000 and has a size of 0x1.0000.
- PCI I/O space that begins at 0x0100.0000 and has a size of 0x3e80.0000.
- A 32-bit, PCI memory space that begins at 0x0000.0000 and has a size of 0x3f00.0000.

The `"ranges"` property for the PCI device would contain the encoded form of the following sequence of numbers:

Table 22 Child-Parent Address Relationships for a PCI Node in a PPCRP Machine

Function	**Child Address**			**Parent Address**	**Size**	
	phys.hi	phys.mid	phys.lo		size.hi	size.lo
SIO	0000.0000	0000.0000	0000.0000	8080.0800	0000.0000	0000.0800
SCSI	0000.0800	0000.0000	0000.0000	8080.1000	0000.0000	0000.0800
Slot A	0000.1000	0000.0000	0000.0000	8080.2000	0000.0000	0000.0800
Slot B	0000.1800	0000.0000	0000.0000	8080.4000	0000.0000	0000.0800
Slot C	0000.2000	0000.0000	0000.0000	8080.8000	0000.0000	0000.0800
ISA I/O space	0100.0000	0000.0000	0000.0000	8000.0000	0000.0000	0001.0000
PCI I/O space	0100.0000	0000.0000	0100.0000	8100.0000	0000.0000	3e80.0000
PCI Memory space	0200.0000	0000.0000	0000.0000	c000.0000	0000.0000	3f00.0000

Here the *phys.hi* component of the child address represents the type of address space and the PCI device numbers, and Bit 31 of the parent address represents "I/O space." (Please see the *PCI Bus Binding to IEEE Standard 1275-1994* for a detailed description of the encoding of the *phys.hi* field.)

The code to create this `"ranges"` property is:

```
\ SIO Configuration Space
0000.0000 encode-int encode+ 0 encode-int encode+ 0 encode-int encode+
8080.0800 encode-int encode+
0 encode-int encode+ 800 encode-int encode+

\ SCSI Configuration Space
0000.0800 encode-int encode+ 0 encode-int encode+ 0 encode-int encode+
8080.1000 encode-int encode+
0 encode-int encode+ 800 encode-int encode+

\ Slot A Configuration Space
0000.1000 encode-int encode+ 0 encode-int encode+ 0 encode-int encode+
8080.2000 encode-int encode+
0 encode-int encode+ 800 encode-int encode+

\ Slot B Configuration Space
0000.1800 encode-int encode+ 0 encode-int encode+ 0 encode-int encode+
8080.4000 encode-int encode+
0 encode-int encode+ 800 encode-int encode+
\ Slot C Configuration Space
0000.2000 encode-int encode+ 0 encode-int encode+ 0 encode-int encode+
8080.8000 encode-int encode+
0 encode-int encode+ 800 encode-int encode+

\ ISA I/O space
0100.0000 encode-int encode+ 0 encode-int encode+ 0 encode-int encode+
8000.0000 encode-int encode+
0 encode-int encode+ 1.0000 encode-int encode+

\ PCI I/O space
0100.0000 encode-int encode+ 0 encode-int encode+ 100.0000 encode-int
encode+
8100.0000 encode-int encode+
0 encode-int encode+ 3e80.0000 encode-int encode+

\ PCI Memory space
0200.0000 encode-int encode+ 0 encode-int encode+ 0 encode-int encode+
c000.0000 encode-int encode+
0 encode-int encode+ 3f00.0000 encode-int encode+
" ranges" property
```

If `"ranges"` exists but its value is of 0 length, the bus's child address space is identical to its parent address space.

If the `"ranges"` property for a particular bus device node is nonexistent, code using that device should use an appropriate default interpretation. Some examples include the following:

- Root node: The root node has no parent. Therefore the correspondence between its child and parent address spaces is meaningless, and there is no need for `"ranges"`.
- SCSI host adapter node: The child address space is not directly addressable, thus `"ranges"` would be meaningless.
- For memory-mapped bus devices where a `"ranges"` property would be meaningful, the absence of a `"ranges"` property is conventionally interpreted to mean that the parent and child address spaces are identical.

The distinction between `"reg"` and `"ranges"` is as follows:

- `"reg"` represents the actual device registers in the parent address space. For a bus adapter, this would be such as configuration/mode/initialization registers.
- `"ranges"` represents the correspondence between a bus adapter's child and parent address spaces.

Most packages do not need to be concerned with `"ranges"`. This property is mainly used for bus bridges. The firmware system does not itself use the `"ranges"` property. `"ranges"` is mainly used by operating systems that wish to auto-configure themselves.

See also: Chapter 11 "Memory-Mapped Buses".

"**reg**"

This property defines the device's addressable regions in its parent's address space.

This property is mandatory for PCI Child Nodes, as defined by *IEEE Standard 1275-1994*. The property value consists of a sequence of (*phys-addr, size*) pairs. In the first such pair, the *phys-addr* component is the Configuration Space address of the beginning of the function's set of configuration registers and the size component is zero. Each additional (*phys-addr, size*) pair specifies the address and characteristics of an addressable region of Memory Space or I/O Space associated with the function including the PCI Expansion ROM.

For a PCI device, the order of the pairs should be:

- An entry describing the Configuration Space for the device.
- An entry for each active base address register (BAR), in Configuration Space order, describing the entire space mapped by that BAR.
- An entry describing the Expansion ROM BAR, if the device has an Expansion ROM.
- An entry for each non-relocatable addressable resource.

In the event that a function has an addressable region that can be accessed relative to more than one Base Address Register (for example, in Memory Space relative to one Base Register, and in I/O Space relative to another), only the primary access path (typically, the one in Memory Space) is listed in the `"reg"` property, and the secondary access path is listed in the `"alternate-reg"` property.

The property value consists of one or more (*phys-addr size*) pairs. For PCI, *phys-addr* is (*phys.lo phys.mid phys.hi*), encoded with `encode-phys`, and *size* is a pair of integers, each encoded with `encode-int`. The first integer denotes the most-significant 32 bits of the 64-bit region size, and the second integer denotes the least-significant 32 bits thereof.

For example, to declare a PCI device with:

- A register field of size 0x100 in 32-bit memory space that is controlled by the first 32-bit base address register.
- A register field of size 0x380 in I/O space that is controlled by the second 32-bit base address register. The register field of interest is offset from the base address register by 0x20.0000.
- A 128Kbyte PCI Expansion ROM.
- A non-relocatable field at 0-fff in I/O space.

use the following:

```
hex
my-address my-space encode-phys                    \ Config space regs
0 encode-int encode+ 0 encode-int encode+
0 0 my-space 0200.0010 or encode-phys encode+     \ Memory space
0 encode-int encode+ 100 encode-int encode+       \ BAR at 0x10
20.0000 0 my-space 0100.0014 or encode-phys       \ I/O space
encode+                                            \ BAR at 0x14
0 encode-int encode+ 380 encode-int encode+

0 0 my-space 0200.0030 or encode-phys encode+     \ PCI Expansion ROM
0 encode-int encode+ 2.0000 encode-int encode+    \ memory space
0 0 my-space 8100.0000 or encode-phys encode+     \ Non-relocatable
0 encode-int encode+ 1000 encode-int encode+      \ memory space
" reg" property
```

In some non-PCI cases, the `reg` command may also be used to create this property. However, `reg` may only be used on buses for which `#size-cells` is one and only when a single `"reg"` property component is required. Consequently, `reg` is never used with PCI devices which require at least three `"reg"` property component (i.e. one component for the card's Configuration Space registers, at least one for the device's functional registers and one for the PCI Expansion ROM).

Note – The contents of the `"reg"` property are used by Open Firmware to determine how large a portion of the system's virtual address space to reserve for use by the card. It is important that the *size* arguments be as large as the actual available addressable resource. If the *size* argument for a region were to be declared smaller than that actually available, and if the driver or a user were to later add a legitimate offset that was larger than *size* to the base address of the region, the resulting virtual address might be within the virtual address space of another card.

See the *PCI Bus Binding to IEEE Standard 1275-1994* for the encoding details.

See also: `reg`, `property`

"revision-id"

This property contains the value of the "Revision ID" register from the Configuration Space header. That register specifies a device-specific revision identifier that is chosen by the vendor. Zero is an acceptable value.

The property value is the register's value encoded with `encode-int`.

See also: *PCI Local Bus Specification*

"slot-names"

This property describes the external labeling of plug-in slots.

The property value is an integer, encoded with `encode-int`, followed by a list of strings, each encoded with `encode-string`.

The integer portion of the property value is a bit mask of available slots; for each add-in slot on the bus, the bit corresponding to that slot's Device Number is set. The least-significant bit corresponds to Device Number 0, the next bit corresponds to Device Number 1, etc. The number of following strings is the same as the number of slots; the first string gives the label that is printed on the chassis for the slot with the smallest Device Number, and so on.

"status"

This optional property indicates the operational status of the device.

Absence of this property means that the operational status of the device is unknown or okay.

If this property is present, the value is a string indicating the status of the device, as follows:

Table 23 `"status"` Property Values

Status Value	Meaning
"okay"	The device is believed to be operational.

Table 23 "status" Property Values *(Continued)*

Status Value	Meaning
"disabled"	The device represented by this node is not operational, but it might become operational in the future (e.g. an external switch is turned off, or something isn't plugged in).
"fail"	The device represented by this node is not operational because a fault has been detected, and it is unlikely that the device will become operational without repair. No additional failure details are available.
"fail-xxx"	The device represented by this node is not operational because a fault has been detected, and it is unlikely that the device will become operational without repair. "xxx" is additional human-readable information about the particular fault condition that was detected.

Used as:

```
" disabled" encode-string  " status" property
```

See also: `property`.

"translations"

This property contains an array of (*phys-addr, virt-addr, size*) entries describing the address translations currently in use by Open Firmware. Those OSs desiring to use Open Firmware services while taking over the memory management function must create all of the translations described by this property's value.

"vendor-id"

This property contains the value of the "Vendor ID" register from the Configuration Space header. That register identifies the manufacturer of the device. Vendor identifiers are assigned by the PCI SIG to ensure uniqueness. 0xffff is an invalid value for vendor-id.

The property value is the register's value encoded with `encode-int`.

See also: *PCI Local Bus Specification*

"width"

This property is associated with `"display"` devices. Encoded with `encode-int`, the property value specifies the number of displayable pixels in the "x" direction of the display.

Manipulating Properties

Property Creation and Modification

The FCode Function `property` is the most general means for creating new properties or modifying the values of existing properties.

There are some special property publishing FCodes, designed for use in common situations:

- `device-name` is a short-hand way to create the "`name`" property.
- `model` is a short-hand way to create the "`model`" property.

- `reg` is a short-hand way to create a "`reg`" property that describes where the package's physical resources are located.

Note – The `reg` method is not useful in a PCI environment since the `"reg"` property for a PCI device will contain information about configuration space, I/O and/or memory space, and the PCI Expansion ROM.

- `delete-property` completely removes a property.

Property Values

Various kinds of information can be stored in a property value byte array by using property encoding and decoding methods. The encoding format is machine-independent; the representation of the property values is independent of the byte organization and word alignment characteristics of any particular processor.

The data type of any particular property must be implicitly known by any software that wishes to use it. In other words, property value data types are not self-identifying. Furthermore, the presence or absence of a property with a particular name can encode a true/false flag; such a property may have a zero-length property value.

Property Encoding

There are three FCodes for encoding a basic piece of data into a property value and one FCode for concatenating the basic pieces for a property that has multiple values.

- `encode-int` encodes a number
- `encode-string` encodes a string
- `encode-bytes` encodes a sequence of bytes
- `encode+` is used to concatenate two previously encoded, basic pieces of data.
- `encode-phys` is an FCode that encodes a physical address (hiding all the relative addressing information). `encode-phys` is derived from `encode-int` and `encode+`.

Property Retrieval

There are three property value retrieving words, `get-my-property`, `get-inherited-property`, and `get-package-property`.

- Use `get-my-property` if the property desired already exists for the package being defined.
- Use `get-package-property` if the property exists in some other package. In this case, you must first find the phandle of the other package, perhaps by using `find-package`.
- Use `get-inherited-property` if the property in question is one that exists somewhere in the chain of parent instances between the package being defined and the root node of the machine. (Using `get-inherited-property` is usually a bad idea because you don't know who supplied the data.)

FCode Programs do not often need to retrieve property values. Such programs usually know the values of their own properties implicitly, and often interact with their parents by calling well-known parent methods.

For an example, suppose a particular PCI FCode package wants to use DVMA to transfer some data between a device and memory.

It could use `my-parent ihandle>phandle get-package-property` to find the value of a property named `slave-only`. `slave-only` will be a property of the parent node of the package being defined, if it exists.

The value of the property is a bit mask of the PCI slots that do *not* support DVMA. Then the package would look at `my-unit` or `my-space` to get its slot number. The two pieces of information will tell the package whether or not it can use DVMA.

Property Decoding

Once a package has found the value of a property of interest, it must decode the value to forms it can understand. If the value is the representation of an integer, use `decode-int` to generate the actual number as a binary number on the stack. If the value of interest is the representation of a string, use `decode-string`. Both of these FCodes act as parsers — they will also return the unused portion of the value for further decoding.

Other kinds of values can be decoded by `left-parse-string` or package-specific decoders. Note that the package must know how to decode the value of a property it wishes to use.

There is no `decode-bytes` function, but it is easy to synthesize if you need it.

```
: decode-bytes  ( addr1 len1 #bytes -- addr len2 addr1 #bytes )
   tuck -      ( addr1 #bytes len2 )
   >r 2dup +   ( addr1 #bytes addr2 ) ( R: len2 )
   r> 2swap
;
```

Property-Specific FCodes

Following is a summary of property-specific FCodes. See the individual dictionary entries in Chapter 12 “Open Firmware Dictionary” for more information.

Table 24 Property-specific FCodes

Name	Stack Comment	Description
Property Creation/Destruction		
`property`	(*prop-addr prop-len name-addr name-len* --)	Create a property named *name-addr name-len* with the value *prop-addr prop-len*.
`device-type`	(*addr len* --)	Shorthand word to create the `"device_type"` property with the value *addr len*.
`model`	(*addr len* --)	Shorthand word to create the `"model"` property with the value *addr len*.
`device-name`	(*addr len* --)	Shorthand macro to create the `"name"` property with the value *addr len*.
`reg`	(*phys.lo … phys.hi size* --)	Shorthand word to create the `"reg"` property.
`delete-property`	(*name-addr name-len* --)	Delete the specified property.
Property Encoding		
`encode-int`	(*n* -- *prop-addr prop-len*)	Converts an integer to a prop-encoded-array.
`encode-phys`	(*phys.lo … phys.hi* -- *prop-addr prop-len*)	Converts a physical unit pair to a prop-encoded-array.
`encode-string`	(*addr len* -- *prop-addr prop-len*)	Converts a text string to a prop-encoded-array.
`encode+`	(*prop-addr1 prop-len1 prop-addr2 prop-len2* -- *prop-addr prop-len1+2*)	Concatenate two prop-encoded-array structures. They must have been created sequentially.
`encode-bytes`	(*addr len* -- *prop-addr prop-len*)	Converts a byte array to a prop-encoded-array. Similar to `encode-string`, except no trailing null is appended.

Table 24 Property-specific FCodes *(Continued)*

Name	Stack Comment	Description
Property Decoding		
`decode-bytes`	(*prop-addr prop-len data-len* -- *prop-addr2 prop-len2 data-addr data-len*)	Decodes *data-len* bytes from the start of a prop-encoded-array returning the byte array and the rest of the prop-encoded-array.
`decode-int`	(*prop-addr prop-len* -- *prop-addr2 prop-len2 n*)	Decodes an integer from the start of a prop-encoded-array returning the integer and the remainder of the prop-encoded-array.
`decode-phys`	(*prop-addr prop-len* -- *prop-addr2 prop-len2 phys.lo … phys.hi*)	Decodes a unit address from the start of a prop-encoded-array returning the address and the rest of the prop-encoded-array.
`decode-string`	(*prop-addr prop-len* -- *prop-addr2 prop-len2 str len*)	Decodes a string from the start of a prop-encoded-array returning the string and the remainder of the prop-encoded-array.
Property Retrieval		
`get-my-property`	(*name-addr name-len* -- *true* | *prop-addr prop-len false*)	Returns the prop-encoded-array contents for the property *addr len* within the current instance, or *true* if not found.
`get-package-property`	(*addr len phandle* -- *true* | *prop-addr prop-len false*)	Returns the prop-encoded-array contents for the property *addr len* within the package *phandle*, or *true* if not found.
`get-inherited-property`	(*addr len* -- *true* | *prop-addr prop-len false*)	Returns the prop-encoded-array contents for the property *addr len*, or *true* if not found. The current package instance is searched first. If not found, the parent is searched next, then the parent's parent, and so on.

6

FCode Basic Concepts

This chapter contains information about a number of FCode concepts that apply to FCode drivers in general. Before reading the chapter(s) devoted to the specific device type(s) of interest to you, please review the information in this chapter. It provides a context for understanding the basic structure and function of an FCode driver.

Parent-Relative Addressing

One of the most powerful concepts of Open Firmware is that of parent-relative addressing. This concept simply means that any given device in the system is only required to understand addresses in terms of its own parent's address space. (And the parent is only required to understand the address space of his parent, and so on.) This concept is a key to FCode portability across systems having vastly different bus topologies.

To support this concept, Open Firmware makes the following provisions:

- A device can only know its address within a system by asking its parent.

  The methods `my-address` and `my-space` are provided for this purpose. The phrase `my-address my-space` always returns *phys.lo … phys.hi* where the total number of cells returned is specified by the value of the device's parent's `#address-cells` property. `my-space` always returns the *phys.hi* cell of a physical address. `my-address` always returns the remaining cells *phys.lo …* .

  For any given bus, the detailed description of a physical address is given in the Open Firmware bus binding document. For the PCI bus, `#address-cells` is 3, and a physical address consists of *phys.lo phys.mid phys.hi*. (See *PCI Bus Binding to IEEE Standard 1275-1994* for more details.)

- A device must ask its parent to translate a physical address known to the device into a virtual address that can be used by the CPU for accessing device resources.

  As will be seen in "Open Firmware Memory Types" on page 93, a number of mapping methods are provided by a parent that enable a child device to request that such a translation be performed on behalf of the device.

- A device must be able to identify its parent by ihandle and must be able to invoke parental methods.

A device’s instance record contains a `value` named `my-parent` which contains the ihandle of the device’s parent.

A child may invoke a parental method with the phrase `my-parent $call-method`. To simplify this process, the method `$call-parent` is provided which is equivalent to the phrase `my-parent $call-method`.

PCI Configuration Space

PCI Configuration Space is a 256 byte region on each PCI device that is used primarily during device initialization. Configuration space must be accessed with special “configuration read” and “configuration write” bus cycles. A more complete description can be found in *PCI Local Bus Specification, Revision 2.1* (or later).

The *PCI Bus Binding to IEEE Standard 1275-1994* requires that a `/pci` bus node provide the following family of access methods for configuration space.

- `config-b@  ( config-addr -- byte )`
- `config-w@  ( config-addr -- word )`
- `config-l@  ( config-addr -- long )`
- `config-b!  ( byte config-addr -- )`
- `config-w!  ( word config-addr -- )`
- `config-l!  ( long config-addr -- )`

The *config-addr* argument represents the address in configuration space of the desired memory location. The address of the first location of a device’s configuration space is the value returned by `my-space` (i.e. *phys.hi*). The addresses of the remaining locations are calculated by adding offsets to the value returned by `my-space`.

Since these are methods of `/pci`, `$call-parent` is typically used in a device driver to invoke these methods as shown in the following code fragment.

```
\ Enable PCI I/O space accesses.
4 constant cmd-reg-offset
1 constant io-space-enable

my-space cmd-reg-offset + dup   ( cmd-addr cmd-addr )
" config-w@" $call-parent       ( cmd-addr cmd-val  )
io-space-enable or              ( cmd-addr cmd-val’ )
swap " config-w!" $call-parent  (   )
```

PCI Configuration Space Header

Figure 5 shows the layout of the configuration space header used by PCI peripheral devices.

Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Revision ID	08h
BIST	Header Type	Latency Timer	Cache Line Size	0Ch
Base Address Registers				10h
				14h
				18h
				1Ch
				20h
				24h
Cardbus CIS Pointer				28h
Subsystem ID		Subsystem Vendor ID		2Ch
Expansion ROM Base Address				30h
Reserved				34h
Reserved				38h
Max_Lat	Min_Gnt	Interrupt Pin	Interrupt Line	3Ch

Figure 5 PCI Configuration Space Header Type 00h

Device ID / Vendor ID

The Open Firmware FCode probing process for the PCI bus uses the values returned for the “device ID” and “vendor ID” fields to differentiate empty and non-empty slots.

- A returned value of 0xFFFFFFFF indicates an empty slot.
- All other values indicate a card present.

On PR*P and CHRP machines, the PCI configuration space header for Slot *N* is located at Address *N**0x800. The following code scans the 32 possible PCI slot numbers on such a machine and prints a formatted listing of the device ID / vendor ID for each slot.

```
hex
dev /pci
20 0 do
   i 3 u.r  i 800 * config-l@ 9 u.r
loop
device-end
```

Command Register

In an FCode driver context, the “command” register is primarily used to enable / disable accesses to the card’s various address spaces and to control whether the device is enabled to act as a bus master.

Bit	Description
0	When set, enables card’s response to I / O space accesses. State after PCI bus reset is 0.
1	When set, enables card’s response to memory space accesses. State after PCI bus reset is 0.
2	When set, enables card’s ability to act as a PCI bus master. State after PCI bus reset is 0.

The following code fragment enables memory space accesses.

```
\ Enable PCI memory space accesses.
4 constant cmd-reg-offset
2 constant memory-space-enable

my-space cmd-reg-offset + dup   ( cmd-addr cmd-addr )
" config-w@" $call-parent       ( cmd-addr cmd-val  )
memory-space-enable or          ( cmd-addr cmd-val’ )
swap " config-w!" $call-parent  (   )
```

Base Address Registers

The “base address” registers enable the relocation of a card’s addressable resources. Bit 0 is read-only and indicates whether a given base address register maps a memory space resource or an I / O space resource.

- The value 0 indicates a memory space resource.
- The value 1 indicates an I / O space resource.

For memory space, the base address register (BAR) bits have the following meanings:

Bit	Description
0	0
1 - 2	00 - Locate anywhere in 32-bit memory space 01 - Locate below 1 MB 10 - Locate anywhere in 64-bit memory space 11 - Reserved
3	Set to 1 if region is prefetchable
4 - 31	Base address

For I/O space, the bits of a base address register have the following meanings:

Bit	Description
0	1
1	Reserved. Must return 0.
2 - 31	Base address

The size of the addressable resource associated with a given BAR can be determined by writing 0xFFFFFFFF to the BAR, reading back the result and interpreting the returned value. In an FCode/Open Firmware context, the system firmware is responsible for determining the size of the associated region and for making the virtual address assignments.

The FCode driver is not involved in the assignment of virtual addresses to the base address registers. In fact, drivers should not ever explicitly access the address values in the base address registers or cache them for later use. Doing so can cause malfunctions of the device driver since there is no guarantee that the value of the base address register will be constant over a given power-cycle of the host machine.

FCode drivers should only deal with the values in the base address registers through the various mapping methods provided by Open Firmware. (These methods and their use in a PCI context will be discussed in detail in "Open Firmware Memory Types.) In a PCI context, the arguments of these methods need:

- The register number of the associated BAR (passed in *phys.hi*)
- The offset, if any, of the region of interest from the base address of the space (passed in *phys.lo phys.mid*).

At probe time, the base address register has not been set to its permanent value. A mapping request performed at probe time results in the base address register being loaded with a temporary value for use at probe time. Final base address register assignments are almost guaranteed to be different from any probe time assignments that may have been made.

Open Firmware Memory Types

From the perspective of an FCode programmer, Open Firmware-based systems have four different types of memory. The following sections describe each of these types and how they are used.

For the purposes of the following discussions, please refer to Figure 6 which shows a generalized system including an expansion bus and a plug-in peripheral card.

System Memory

The term "system memory" refers to the host system's CPU memory irrespective of whether there is a memory cache involved. System memory is generally obtained with the data structure creation methods `constant`, `value`, variable or buffer: . Accesses to system memory have no "side effects" (i.e. no other system state changes as a result).

FCode programs access this memory with the `@` and `!` families of methods (i.e. `@`, `c@`, `w@`, `l@` and `x@`, and `!`, `c!`, `w!`, `l!` and `x!`). These methods are not required to be atomic

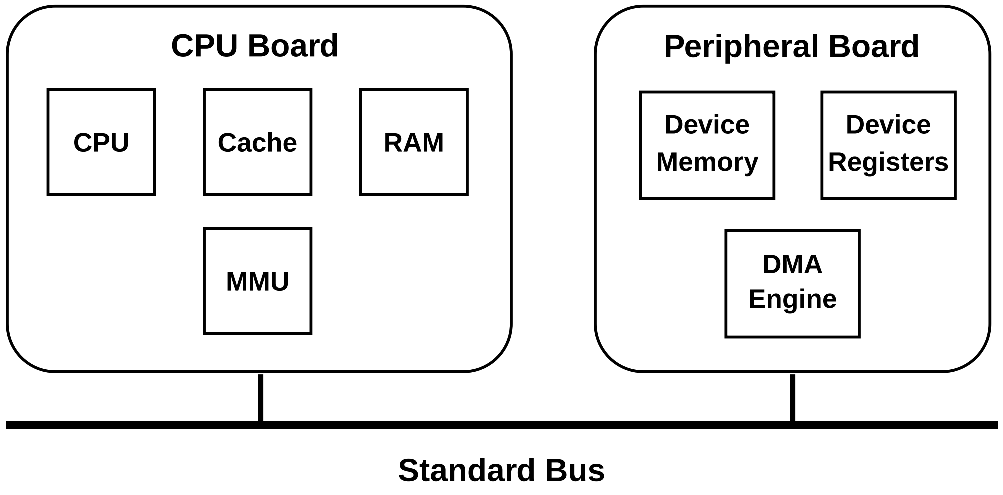

Figure 6 Hypothetical System with Plug-in Peripheral Card

(e.g. an Open Firmware implementation can choose to do multi-byte fetches and stores as a series of byte fetches and stores).

Scratch Buffer

The term "scratch buffer" refers to a region of system memory that is acquired for use with `alloc-mem` and is returned after use with `free-mem`.

As with "system memory", accesses to a scratch buffer has no side effects, and FCode programs access this memory with the `@` and `!` families of methods.

For example:

```
50 constant buf-len
-1 value buf-addr          \ Conventionally set to -1 when invalid

buf-len alloc-mem          \ Obtain memory buffer and virtual address
                           \ of first location in buffer
to buf-addr                \ Store virtual address
...
h# 1234 buf-addr 2+ w!     \ Write 1234 to second word in buffer
...
buf-addr buf-len free-mem  \ Free buffer and return virtual address
```

DMA Memory

The term "DMA memory" refers to regions of system memory that can be accessed both by the system's CPU and by a DMA engine on a peripheral card (i.e. a "bus master" card) on an expansion bus.

From the perspective of the FCode driver, this memory appears in the memory space of the device's parent. Thus, DMA memory is directly accessible by the device's DMA engine and is usually directly accessible by the CPU.

It is very important to note that, in the general case, the addresses used by the DMA engine and by the CPU are not necessarily the same address. The DMA address is the address that appears on the bus to which the bus master device is directly connected. In general, the bridges between that DMA bus and the system's memory bus may perform some address translation. The address translation through those bridges in the direction from CPU to memory is not necessarily the same as the translation in the direction from bus master device to memory. One source of confusion is the fact that, in some systems and for some buses, the translation may just *happen* to be the same in some pairs of cases. If you get used to writing drivers for systems where this is the case, it often comes as a surprise when you must make the distinction.

Two methods are used to obtain DMA memory and to create the appropriate address translations for the use of that memory by the CPU and the bus master device.

- `dma-alloc` allocates a region of physical memory suitable for use with DMA and returns a virtual address for the CPU's use.
- `dma-map-in` converts that CPU virtual address into a DMA physical address suitable for use by the device's DMA engine.

Since some systems include a memory cache as shown in Figure 6, the information stored in the DMA memory and in the cache may not be identical until the cache is flushed. The `dma-sync` method is provided for this purpose and should always be included in an FCode driver. On those systems that do not include a cache, `dma-sync` will be a no-op and so will cause no problems. Failure to include `dma-sync` will cause a driver to fail if it is ever used in a system that includes a cache.

Memory obtained with `dma-alloc` must be freed with `dma-free`. In addition, DMA mappings created with `dma-map-in` are destroyed with dma-map-out.

An FCode driver will normally use the various DMA memory management methods of its parent. For example:

```
50 constant dma-buf-len
-1 value cpu-addr
-1 value dev-addr

dma-buf-len " dma-alloc" $call-parent  to cpu-addr
cpu-addr dma-buf-len true " dma-map-in" $call-parent  to dev-addr
...
cpu-addr dev-addr dma-buf-len " dma-sync" $call-parent
...
cpu-addr dev-addr dma-buf-len " dma-map-out" $call-parent
cpu-addr dma-buf-len " dma-free" $call-parent
```

In summary, the general algorithm for using DMA memory is:

- The required DMA-accessible memory is obtained with `dma-alloc` which allocates the requested memory and returns the virtual address used by the CPU.
- `dma-map-in` is used to translate the CPU virtual address into a physical address for use by the bus master device.
- In the case of moving data to the device:
  - The data is written into the DMA memory by the CPU using its virtual address and `dma-sync` is invoked to flush any cache that might be present.
  - The DMA engine controller is set up using the address returned by `dma-map-in` and the DMA process is started.

- In the case of moving data from the device:
  - The DMA engine controller is set up and started. When the DMA process completes, `dma-sync` is invoked to flush any cache that might be present.
  - The data is read from the memory by the CPU.
- When all of the DMA operations are complete, `dma-map-out` is invoked to return the DMA physical address. `dma-free` is invoked to free the allocated DMA memory and to return the CPU virtual address.

Device Memory

The term “device memory” refers to memory and/or registers located on a peripheral card. Since the address of this memory is known to the driver in terms of a physical address in the device’s parent’s address space, the `map-in` method is used to convert such a physical address into a virtual address suitable for use by the CPU. `map-out` is used to return the virtual address when the address is no longer needed.

Accesses to device memory may have “side effects”. For example, the reading of an interrupt status register may affect the contents of that register. Consequently, there are special families of `@` and `!` methods for accessing device memory. These methods, `rb@`, `rw@`, `rl@` and `rx@`, and `rb!`, `rw!`, `rl!` and `rx!`, are guaranteed to be atomic (e.g. Open Firmware ensures that the access resulting from the use of one of these methods is complete before any other method is allowed to be executed).

PCI Device Register Mapping and Use

The following examples shown how to map several different styles of device memory on a PCI device. Refer to *PCI Bus Binding to IEEE Standard 1275-1994* for a detailed explanation of the various bits in the *phys.lo … phys.hi* arguments.

The first example shows how to map relocatable memory in 32-bit memory space. The example assumes that the region of interest is 8 bytes in size and is associated with the base address register located at offset 0x10 in the PCI Configuration Space header.

```
8 constant /mem-regs      \ Device has 8 byte-wide registers
-1 value   reg-mem-addr   \ Storage to hold virtual address

my-address my-space       ( phys.lo phys.mid phys.hi )
\ Modify phys.hi to indicate relocatable 32-bit memory space using the
\ BAR @ offset h# 10
0200.0010 or              ( phys.lo phys.mid phys.hi’ )
/mem-regs                 ( phys.lo phys.mid phys.hi’ size )
" map-in" $call-parent    ( virt )
to reg-mem-addr           (      )
...
33 reg-mem-addr 5 + rb!   \ Stores 33 to sixth byte in mapped region
...
reg-mem-addr /mem-regs    ( virt size )
" map-out" $call-parent   (           )
```

The next example shows how to map relocatable memory in I/O space. The example assumes that the region of interest is 16 bytes in size and is offset by 32 bytes from the start of the region associated with the base address register located at offset 0x14 in the PCI Configuration Space header.

```
10 constant /io-regs      \ Device has 16 byte-wide registers
20 constant reg-io-offset \ Offset from start of region described by BAR
-1 value    reg-io-addr   \ Storage to hold virtual address

my-address reg-io-offset 0 d+   ( phys.lo’ phys.mid )
my-space                        ( phys.lo’ phys.mid phys.hi )
\ Modify phys.hi to indicate relocatable I/O space using the BAR @
\ offset h# 14
0100.0014 or                    ( phys.lo’ phys.mid phys.hi’ )
/io-regs                        ( phys.lo’ phys.mid phys.hi’ size )
" map-in" $call-parent          ( virt )
to reg-io-addr                  (      )
...
1234 reg-io-addr e + rw!      \ Stores 1234 to last word in mapped region
...
reg-io-addr /io-regs            ( virt size )
" map-out" $call-parent         (           )
```

The last example shows how to map non-relocatable memory in I/O space. The example assumes that the region of interest is located at absolute address 0xABCD and is 256 bytes in size. There is no associated base address register since this is non-relocatable space.

```
100 constant /io-regs     \ Device has 256 byte-wide registers
-1 value     reg-io-addr  \ Storage to hold virtual address

abcd 0 my-space                 ( phys.lo phys.mid phys.hi )
\ Modify phys.hi to indicate non-relocatable I/O space
8100.0000 or                    ( phys.lo phys.mid phys.hi’ )
/io-regs                        ( phys.lo phys.mid phys.hi’ size )
" map-in" $call-parent          ( virt )
to reg-io-addr                  (      )
...
\ Stores 12345678 to first long word in mapped region
12345678 reg-io-addr rl!
...
reg-io-addr /io-regs            ( virt size )
" map-out" $call-parent         (           )
```

All of the examples above are written using the virtual address pointer directly to calculate the address in which to store data. However, experienced Forth/FCode programmers would “factor” this operation (i.e. create another word which does the address calculation internally). A very common “factoring” in this situation would be to create a family of access words that take an offset as an argument and either fetch or store data to that offset.

For example, if the non-relocatable region mapped in the last example were always accessed as long words, the following two methods would be an appropriate factoring of the code.

```
: iol!  ( data offset -- )
   reg-io-addr + rl!
;
```

```
: iol@  ( offset -- data )
   reg-io-addr + rl@
;
```

Factoring your code will make it easier to write, test and debug. It will also make your FCode image smaller. Most importantly, it takes advantage of the inexpensive context switches made possible by the Forth language and uses them to their best effect.

After you have mapped PCI device memory, you must also enable memory and/or I/O space accesses before your mapping(s) will work correctly. Before your device can act as a bus master, bus mastering must also be enabled. The enable bits for all of these functions are contained in the “command” register of the PCI Configuration Space header. See “Command Register” on page 92 for more details and for a code example of enabling memory space.

When a driver no longer needs access to an address space, it should disable accesses to that space. The following code fragment disables memory space accesses.

```
\ Disable PCI memory space accesses.
4 constant cmd-reg-offset
2 constant memory-space-enable

my-space cmd-reg-offset + dup   ( cmd-addr cmd-addr )
" config-w@" $call-parent       ( cmd-addr cmd-val  )
memory-space-enable not and     ( cmd-addr cmd-val′ )
swap " config-w!" $call-parent  (   )
```

7

Block and Byte Devices

Block Devices

Block devices are nonvolatile mass storage devices whose information can be accessed in any order. Examples of block devices include hard disks, floppy disks, and CD-ROMs. Open Firmware typically uses block devices for booting.

This device type generally applies to disk devices, but as far as Open Firmware is concerned, it simply means that the device “looks like a disk” at the Open Firmware software interface level.

The block device’s FCode must declare the `block` device type, and must implement the methods `open` and `close`, as well as the methods described below in “Required Methods” on page 100.

Although packages of the `block` device type present a byte-oriented interface to the rest of the system, the associated hardware devices are usually block-oriented i.e. the device reads and writes data in “blocks” (groups of, for example, 512 or 2048 bytes). The standard `/deblocker` support package assists in the presentation of a byte-oriented interface “on top of” an underlying block-oriented interface, implementing a layer of buffering that “hides” the underlying “block” length.

Block devices are often subdivided into several logical “partitions”, as defined by a disk label - a special block, usually the first one on the device, which contains information about the device. The driver is responsible for appropriately interpreting a disk label. The driver may use the standard `disk-label` support package if the device does not implement a specialized label. The `disk-label` support package interprets one or more system-dependent label formats. Since the disk booting protocol usually depends upon the label format, the standard `disk-label` support package also implements a `load` method for the corresponding boot protocol.

Byte Devices

Byte devices are sequential-access mass storage devices, for example tape devices. Open Firmware typically uses byte devices for booting.

The byte device’s FCode program must declare the `byte` device type, and must implement the `open` and `close` methods in addition to those described in “Required Methods”.

Although packages of the `byte` device type present a byte-oriented interface to the rest of the system, the associated hardware devices are usually record-oriented; the device reads and writes data in records containing more than one byte. The records may be fixed length or variable length. The standard `deblocker` support package assists in presenting a byte-oriented interface on top of an underlying record-oriented interface, implementing a layer of buffering that hides the underlying record structure.

Required Methods

block-size (-- block-len)

Return the record size *block-len* (in bytes) of all data transfers to or from the device. The most common value for *block-len* is 512.

This method is only required if the `deblocker` support package is used.

dma-alloc (size -- virt)

Allocates *size* bytes of memory, contiguous within the direct-memory-access address space of the device’s bus, suitable for DMA. Returns the virtual address *virt*.

This method is only required if the `deblocker` support package is used.

dma-free (virt size --)

Frees *size* bytes of memory at virtual address *virt* that were previously allocated with `dma-alloc`.

This method is only required if the `deblocker` support package is used.

load (addr -- size)

`load` works a bit differently for block and byte devices:

With block devices, it loads a stand-alone program from the device into memory at *addr*. *size* is the size in bytes of the program loaded. If the device can contain several such programs, the instance arguments returned by `my-args` can be used to select the specific program desired. `open` is executed before `load` is invoked.

With byte devices, `load` reads a stand-alone program from the tape file specified by the value of the argument string given by `my-args`. That value is the string representation of a decimal integer. If the argument string is null, tape file 0 is used. `load` places the program in memory at *addr*, returning the *size* of the read-in program in bytes.

max-transfer (-- max-len)

Return the size in bytes of the largest single transfer that the device can perform. `max-transfer` is expected to be a multiple of `block-size`.

This method is only required if the `deblocker` support package is used.

read (addr len -- actual)

Read at most *len* bytes from the device into memory at *addr*. *actual* is the number of bytes actually read. If the number of bytes read is 0 or negative, the read failed. Note that *len* need not be a multiple of the device’s normal block size.

read-blocks (addr block# #blocks -- #read)

Read *#blocks* records of length `block-size` bytes each from the device, starting at device block *block#*, into memory at address *addr*. *#read* is the number of blocks actually read.

This method is only required if the `deblocker` support package is used.

seek (pos.lo pos.hi -- status) for block; (offset file# -- error?) for byte

`seek` works a bit differently depending on whether it's being used with a block or byte device.

For block devices, `seek` sets the device position for the next read or write. The position is the byte offset from the beginning of the device specified by the 64-bit number which is the concatenation of *poshigh* and *poslow*. *status* is -1 if the seek fails, and 0 or 1 if it succeeds.

For byte devices, it seeks to the byte *offset* within file *file#*. If *offset* and *file#* are both 0, rewind the tape. *error?* is -1 if seek fails, and 0 if seek succeeds.

write (addr len -- actual)

Write *len* bytes from memory at *addr* to the device. *actual* is the number of bytes actually written. If *actual* is less than *len*, the write did not succeed. *len* need not be a multiple of the device's normal block size.

write-blocks (addr block# #blocks -- #written)

Write *#blocks* records of length `block-size` bytes each to the device, starting at block *block#*, from memory at *addr*. *#written* is the number of blocks actually written.

If the device is not capable of random access (e.g. a sequential access tape device), *block#* is ignored.

This method is only required if the `deblocker` support package is used.

Required Properties

Table 25 Required Properties of Block and Byte Devices

Property Name	Sample Value
`name`	`" FirmWorks,googly"`
`reg`	list of registers (device-dependent)
`device_type`	`" block" or " byte"`

Device Driver Examples

The structure of the device tree for the sample card supported by the sample device drivers in this chapter is as follows:

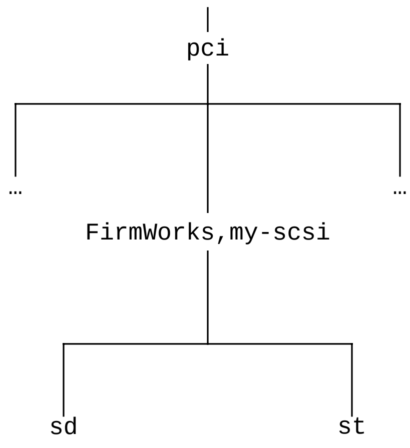

Figure 7 Sample Device Tree

Simple Block Device Driver

Code Example 7-1 Simple Block Device Driver

```
\ This is at a stage where each leaf node can be used only as a non-bootable device.
\ It only creates nodes and publishes necessary properties to identify the device.

fcode-version2
hex
: copyright  ( -- )
   ." Copyright (c) 1994-1996 FirmWorks.  All Rights Reserved." cr
;
" FirmWorks,my-scsi"  encode-string " name" property

h# 20.0000   constant scsi-offset
h# 40        constant /scsi

\ Define "reg" property
\ PCI Configuration Space
my-address my-space encode-phys  0 encode-int encode+  0 encode-int encode+

\ Memory Space Base Address Register 10
my-address scsi-offset 0 d+ my-space 0200.0010 or encode-phys  encode+
0 encode-int encode+  /scsi encode-int encode+

\ PCI Expansion ROM
my-address my-space h# 200.0030 or encode-phys encode+
0 encode-int encode+ h# 10.0000 encode-int encode+
" reg" property

new-device   \ missing "reg" indicates a SCSI "wild-card" node
   " sd"     encode-string " name" property
finish-device
```

Code Example 7-1 Simple Block Device Driver *(Continued)*

```
new-device   \ missing "reg" indicates a SCSI "wild-card" node
    " st"    encode-string " name" property
finish-device
fcode-end
```

Extended Block Device Driver

Code Example 7-2 Sample Driver for "my-scsi" Device

```
\ sample driver for "my-scsi" device
\ It is still a non-bootable device. The purpose is to show how an intermediate stage
\ of driver can be used to debug board during development. In addition to publishing
\ the properties, this sample driver shows methods to access, test and control
\ "FirmWorks,my-scsi" device.

\ The following main methods are provided for "FirmWorks,my-scsi" device.
\  open  ( -- okay? )
\  close  ( -- )
\  reset  ( -- )
\  selftest  ( -- error? )

fcode-version2
hex
headers
: copyright  ( -- )
   ." Copyright (c) 1994-1996 FirmWorks.  All Rights Reserved." cr
;

h# 20.0000    constant scsi-offset
h# 40         constant /scsi
d# 25.000.000 constant clock-frequency
" FirmWorks,my-scsi" device-name

\ Define "reg" property
\ PCI Configuration Space
my-address my-space encode-phys  0 encode-int encode+  0 encode-int encode+

\ Memory Space Base Address Register 10
my-address scsi-offset 0 d+ my-space 0200.0010 or encode-phys  encode+
0 encode-int encode+  /scsi encode-int encode+

\ PCI Expansion ROM
my-address my-space h# 200.0030 or encode-phys encode+
0 encode-int encode+ h# 10.0000 encode-int encode+

" reg" property

\ Configuration register access words
: my-w@ ( offset -- w ) my-space + " config-w@" $call-parent ;
: my-w! ( w offset -- ) my-space + " config-w!" $call-parent ;

h# 10.0000 constant dma-offset
h# 10      constant /dma
-1 instance value dma-chip



\ Methods to access/control DMA registers during development
: dmaaddress  ( -- addr )  dma-chip 4 +  ;
: dmacount    ( -- addr )  dma-chip 8 +  ;
: dmaaddr@    ( -- n )     dmaaddress rl@  ;
: dmaaddr!    ( n -- )     dmaaddress rl!  ;
: dmacount@   ( -- n )     dmacount rl@  ;
: dmacount!   ( n -- )     dmacount rl!  ;
```

Code Example 7-2 Sample Driver for "my-scsi" Device *(Continued)*

```
: dma-chip@   ( -- n )     dma-chip rl@  ;
: dma-chip!   ( n -- )     dma-chip rl!  ;
: dma-btest   ( mask -- flag )  dma-chip@  and  ;
: dma-bset    ( mask -- )       dma-chip@  or  dma-chip!  ;
: dma-breset  ( mask -- )  not dma-btest  dma-chip!  ;

external

\ Methods to allocate, map, unmap, free DMA buffers
: decode-unit  ( addr len -- low high )         decode-2int  ;
: dma-alloc    ( size -- vaddr )                " dma-alloc" $call-parent  ;
: dma-free     ( vaddr size -- )                " dma-free" $call-parent  ;

\ Since the PCI bus uses physical addressing, devaddr returned by dma-map-in is the
\ physical address associated with vaddr.
: dma-map-in   ( vaddr size cache? -- devaddr )  " dma-map-in" $call-parent  ;
: dma-map-out  ( vaddr devaddr size -- )        " dma-map-out" $call-parent  ;

\ dma-sync could be a dummy routine if the parent device doesn't support.
: dma-sync  ( virt-addr dev-addr size -- )
   " dma-sync" my-parent ['] $call-method catch  if
      2drop 2drop 2drop
   then
;

: map-in   ( addr space size -- virt )  " map-in"  $call-parent  ;
: map-out  ( virt size -- )             " map-out" $call-parent  ;

\ Some of Apple’s Open Firmware implementations have a bug in their map-in method. The
\ bug causes phys.lo and phys.mid to be treated as absolute addresses rather than
\ offsets even when working with relocatable addresses.

\ To overcome this bug, the Open Firmware Working Group in conjunction with Apple has
\ adopted a workaround that is keyed to the presence or absence of the add-range method
\ in the PCI node. If the add-range method is present in an Apple ROM, the map-in
\ method is broken. If the add-range property is absent, the map-in method behaves
\ correctly.

\ The following methods allow the FCode driver to accomodate both broken and working
\ map-in methods.

: map-in-broken?  ( -- flag )
   \ Look for the method that is present when the bug is present
   " add-range"  my-parent  ihandle>phandle   ( adr len phandle )
   find-method  dup  if  nip  then            ( flag )  \ Discard xt if present
;




\ Return phys.lo and phys.mid of the address assigned to the PCI base address
\ register indicated by phys.hi .
: get-base-address  ( phys.hi -- phys.lo phys.mid phys.hi )
   " assigned-addresses" get-my-property  if   ( phys.hi )
      ." No address property found!" cr
      0 0 rot  exit                            \ Error exit
   then                      ( phys.hi adr len )
```

Code Example 7-2 Sample Driver for "my-scsi" Device *(Continued)*

```
   rot >r                    ( adr len )  ( r: phys.hi )
   \ Found assigned-addresses, get address
   begin  dup  while         ( adr len' )  \ Loop over entries
      decode-phys            ( adr len' phys.lo phys.mid phys.hi )
      h# ff and  r@ h# ff and  =  if  ( adr len' phys.lo phys.mid )  \ This one?
         2swap 2drop         ( phys.lo phys.mid )          \ This is the one
         r> exit             ( phys.lo phys.mid phys.hi )
      else                   ( adr len' phys.lo phys.mid ) \ Not this one
         2drop               ( adr len' )
      then                   ( adr len' )
      decode-int drop decode-int drop        \ Discard boring fields
   repeat
   2drop                     ( )
   ." Base address not assigned!" cr

   0 0 r>                    ( 0 0 phys.hi )
;

headers

: dma-open   ( -- )
   my-address dma-offset 0 d+  my-space /dma map-in  to dma-chip
;
: dma-close  ( -- )  dma-chip /dma map-out  -1 to dma-chip  ;

-1 instance value scsi-init-id
-1 instance value scsi-chip

h# 20 constant /mbuf
-1 instance value mbuf
-1 instance value mbuf-dma

d# 6 constant /sense
-1 instance value sense-command
-1 instance value sense-cmd-dma

d# 8 constant #sense-bytes
-1 instance value sense-buf
-1 instance value sense-buf-dma
-1 instance value mbuf0

d# 12 constant /cmdbuf
-1 instance value cmdbuf
-1 instance value cmdbuf-dma
-1 instance value scsi-statbuf




\ Mapping and allocation routines for SCSI.
: map-scsi-chip  ( -- )
   map-in-broken?  if
      my-space h# 8200.0010 or  get-base-address   ( phys.lo phys.mid phys.hi )
   else
      my-address my-space h# 200.0010 or           ( phys.lo phys.mid phys.hi )
   then                                            ( phys.lo phys.mid phys.hi )
```

Code Example 7-2 Sample Driver for "my-scsi" Device *(Continued)*

```
   /scsi map-in  to scsi-chip

   4 dup my-w@ 6 or swap my-w!  \ Enable memory space and bus mastering
   scsi-chip encode-int " address" property
;

: unmap-scsi-chip  ( -- )
   4 dup my-w@ 6 invert and swap my-w!  \ Disable memory space and bus mastering
   scsi-chip /scsi map-out  -1 to scsi-chip
   " address" delete-property
;

\ After any changes to sense-command by CPU or any changes to sense-cmd-dma by
\ device, synchronize changes by issuing " sense-command sense-cmd-dma /sense
\ dma-sync " Similarly after any changes to sense-buf, sense-buf-dma, mbuf,
\  mbuf-dma, cmdbuf or cmdbuf-dma, synchronize changes by appropriately issuing
\  dma-sync map scsi chip and allocate buffers for "sense" command and status
: map-scsi  ( -- )
   map-scsi-chip
   /sense dma-alloc to sense-command
   sense-command /sense false dma-map-in  to sense-cmd-dma
   #sense-bytes dma-alloc to sense-buf
   sense-buf #sense-bytes false dma-map-in  to sense-buf-dma
   2 alloc-mem to scsi-statbuf
;
\ free buffers for "sense" command and status and unmap scsi chip
: unmap-scsi  ( -- )
   scsi-statbuf 2 free-mem
   sense-buf sense-buf-dma #sense-bytes dma-sync  \ redundant
   sense-buf sense-buf-dma #sense-bytes dma-map-out
   sense-buf #sense-bytes dma-free
   sense-command sense-cmd-dma /sense dma-sync     \ redundant
   sense-command sense-cmd-dma /sense dma-map-out
   sense-command /sense dma-free
   -1 to sense-command
   -1 to sense-cmd-dma
   -1 to sense-buf
   -1 to scsi-statbuf
   -1 to sense-buf-dma
   unmap-scsi-chip
;

\ constants related to scsi commands
h#  0 constant nop
h#  1 constant flush-fifo
h#  2 constant reset-chip
h#  3 constant reset-scsi
h# 80 constant dma-nop

\ Methods to get SCSI register addresses.
\ Each chip register is one byte, aligned on a 4-byte boundary.
: scsi+  ( offset -- addr )  scsi-chip +  ;
: transfer-count-lo     ( -- addr )  h#  0 scsi+  ;
: transfer-count-hi     ( -- addr )  h#  4 scsi+  ;
: fifo                  ( -- addr )  h#  8 scsi+  ;
: command               ( -- addr )  h#  c scsi+  ;
```

Code Example 7-2 Sample Driver for "my-scsi" Device *(Continued)*

```
: configuration         ( -- addr )  h# 20 scsi+  ;
: scsi-test-reg         ( -- addr )  h# 28 scsi+  ;
\ Read only registers:
: scsi-status           ( -- addr )  h# 10 scsi+  ;
: interrupt-status      ( -- addr )  h# 14 scsi+  ;
: sequence-step         ( -- addr )  h# 18 scsi+  ;
: fifo-flags            ( -- addr )  h# 1c scsi+  ;
\ Write only registers:
: select/reconnect-bus-id  ( -- addr )  h# 10 scsi+  ;
: select/reconnect-timeout ( -- addr )  h# 14 scsi+  ;
: sync-period              ( -- addr )  h# 18 scsi+  ;
: sync-offset              ( -- addr )  h# 1c scsi+  ;
: clock-conversion-factor  ( -- addr )  h# 24 scsi+  ;

\ Methods to read from/store to SCSI registers.
: cnt@      ( -- w )  transfer-count-lo rb@  transfer-count-hi rb@  bwjoin  ;
: fifo@     ( -- c )  fifo rb@  ;
: cmd@      ( -- c )  command rb@  ;
: stat@     ( -- c )  scsi-status rb@  ;
: istat@    ( -- c )  interrupt-status rb@  ;
: fifo-cnt  ( -- c )  fifo-flags rb@  h# 1f and ;
: data@     ( -- c )  begin  fifo-cnt  until  fifo@  ;
: seq@      ( -- c )  sequence-step rb@  h# 7 and ;
: fifo!     ( c -- )  fifo rb!  ;
: cmd!      ( c -- )  command rb!  ;
: cnt!      ( w -- )  wbsplit transfer-count-hi rb! transfer-count-lo rb!  ;
: targ!     ( c -- )  select/reconnect-bus-id rb!  ;
: data!     ( c -- )  begin  fifo-cnt d# 16 <>  until  fifo!  ;

\ SCSI chip NOOP  and initialization
: scsi-nop   ( -- )  nop cmd!  ;
: init-scsi  ( -- )  reset-chip cmd!  scsi-nop  ;

: wait-istat-clear  ( -- error? )
   d# 1000
   begin
      1 ms 1-  ( count )
      dup 0=   ( count expired? )
      istat@   ( count expired? istat )
      0= or    ( count clear? )
   until       ( count )
   0=  if
      istat@ 0<>  if
         cr ." Can't clear ESP interrupts: "
         ." Check SCSI Term. Power Fuse." cr
         true  exit
      then
   then
   false
;
: clk-conv-factor  ( -- n )  clock-frequency d# 5.000.000 / 7 and  ;

\ Initialize SCSI chip, tune time amount, set async operation mode, and set scsi
\ bus id
: reset-my-scsi ( -- error? )
   init-scsi
   h# 93 select/reconnect-timeout rb!
```

Code Example 7-2 Sample Driver for "my-scsi" Device *(Continued)*

```
   0 sync-offset rb!
   clk-conv-factor clock-conversion-factor rb!
   h# 4 scsi-init-id 7 and or  configuration rb!
   wait-istat-clear
;

: reset-bus ( -- error? )
   reset-scsi cmd!  wait-istat-clear
;

: init-n-test  ( -- ok? ) reset-my-scsi 0=  ;

: get-buffers ( -- )
   h# 8000 dma-alloc to mbuf0
   /cmdbuf dma-alloc to cmdbuf
   cmdbuf /cmdbuf false dma-map-in  to cmdbuf-dma
;

: give-buffers ( -- )
   mbuf0 h# 8000 dma-free  -1 to mbuf0
   cmdbuf cmdbuf-dma /cmdbuf dma-sync             \ redundant
   cmdbuf cmdbuf-dma /cmdbuf dma-map-out
   cmdbuf /cmdbuf dma-free
   -1 to cmdbuf  -1 to cmdbuf-dma
;

: scsi-selftest ( -- fail? )  reset-my-scsi  ;

\ dma-alloc and dma-map-in mbuf-dma
: mbuf-alloc ( -- )
   /mbuf dma-alloc  to mbuf
   mbuf /mbuf false dma-map-in  to mbuf-dma
;

\ dma-map-out and dma-free mbuf-dma
: mbuf-free ( -- )
   mbuf mbuf-dma /mbuf dma-sync                \ redundant
   mbuf mbuf-dma /mbuf dma-map-out
   mbuf /mbuf dma-free
   -1 to mbuf
   -1 to mbuf-dma
;

external






\ If any routine were using buffers allocated by dma-alloc, and were using dma mapped
\ by dma-map-in, it would have to dma-sync those buffers after making any changes to
\ them.
: open  ( -- success? )
   dma-open
   " scsi-initiator-id" get-inherited-property 0=  if
      decode-int  to scsi-init-id
```

Code Example 7-2 Sample Driver for "my-scsi" Device *(Continued)*

```
      2drop
      map-scsi
      init-n-test                   ( ok? )
      dup if                        ( true )
         get-buffers                ( true )
      else
         unmap-scsi dma-close       ( false )
      then                          ( success? )
   else
      ." Missing initiator id" cr  false
      dma-close
   then                             ( success? )
;
: close  ( -- )
   give-buffers unmap-scsi dma-close
;

: reset  ( -- )
   dma-open map-scsi
   h# 80 dma-breset
   reset-my-scsi drop reset-bus drop
   unmap-scsi dma-close
;

\ If scsi-selftest were actually using buffers allocated by mbuf-alloc, it would
\ have to do dma-sync after any changes to mbuf or mbuf-dma.
: selftest  ( -- fail? )
   map-scsi
   mbuf-alloc
   scsi-selftest
   mbuf-free
   unmap-scsi
;
new-device  \ missing "reg" indicates a SCSI "wild-card" node
   " sd"     encode-string " name" property
finish-device

new-device  \ missing "reg" indicates a SCSI "wild-card" node
   " st"     encode-string " name" property
finish-device
fcode-end
```

Complete Block and Byte Device Driver

Code Example 7-3 Sample Driver for Bootable Devices

```
\ Sample bootable block and byte device driver
\ This driver supports "block" and "byte" type bootable devices, by using standard
\ "deblocker"and "disk-label" packages.

\ The following main methods are provided for "FirmWorks,my-scsi" device.
\  open  ( -- okay? )
\  close  ( -- )
\  reset  ( -- )
\  selftest  ( -- error? )

fcode-version2
hex
headers
: copyright  ( -- )
   ." Copyright (c) 1994-1996 FirmWorks.  All Rights Reserved." cr
;
h# 20.0000    constant scsi-offset
h# 40         constant /scsi
d# 25.000.000 constant clock-frequency
h# 10.0000    constant dma-offset
h# 10         constant /dma
-1 instance value dma-chip

" FirmWorks,my-scsi" device-name
" scsi" device-type

\ Define "reg" property
\ PCI Configuration Space
my-address my-space encode-phys  0 encode-int encode+  0 encode-int encode+

\ Memory Space Base Address Register 10
my-address scsi-offset 0 d+ my-space 0200.0010 or encode-phys  encode+
0 encode-int encode+  /scsi encode-int encode+

\ PCI Expansion ROM
my-address my-space h# 200.0030 or encode-phys encode+
0 encode-int encode+ h# 10.0000 encode-int encode+
" reg" property

\ Configuration register access words
: my-w@ ( offset -- w ) my-space + " config-w@" $call-parent ;
: my-w! ( w offset -- ) my-space + " config-w!" $call-parent ;

external

\ Methods to allocate, map, unmap, free DMA buffers
: decode-unit  ( addr len -- low high )         decode-2int  ;
: dma-alloc    ( size -- vaddr )                " dma-alloc" $call-parent  ;
: dma-free     ( vaddr size -- )                " dma-free" $call-parent  ;

\ Since the PCI bus uses physical addressing, devaddr returned by dma-map-in is the
\ physical address associated with vaddr.
: dma-map-in   ( vaddr size cache? -- devaddr )  " dma-map-in" $call-parent  ;
: dma-map-out  ( vaddr devaddr size -- )        " dma-map-out" $call-parent  ;
```

Code Example 7-3 Sample Driver for Bootable Devices *(Continued)*

```
\ dma-sync could be dummy routine if parent device doesn't support.
: dma-sync  ( virt-addr dev-addr size -- )
   " dma-sync" my-parent ['] $call-method catch  if
   2drop 2drop 2drop
then
;
: map-in   ( addr space size -- virt )  " map-in"  $call-parent  ;
: map-out  ( virt size -- )             " map-out" $call-parent  ;

\ Some of Apple’s Open Firmware implementations have a bug in their map-in method. The
\ bug causes phys.lo and phys.mid to be treated as absolute addresses rather than
\ offsets even when working with relocatable addresses.

\ To overcome this bug, the Open Firmware Working Group in conjunction with Apple has
\ adopted a workaround that is keyed to the presence or absence of the add-range method
\ in the PCI node. If the add-range method is present in an Apple ROM, the map-in
\ method is broken. If the add-range property is absent, the map-in method behaves
\ correctly.

\ The following methods allow the FCode driver to accomodate both broken and working
\ map-in methods.

: map-in-broken?  ( -- flag )
   \ Look for the method that is present when the bug is present
   " add-range"  my-parent  ihandle>phandle   ( adr len phandle )
   find-method  dup  if  nip  then            ( flag )  \ Discard xt if present
;

\ Return phys.lo and phys.mid of the address assigned to the PCI base address
\ register indicated by phys.hi .
: get-base-address  ( phys.hi -- phys.lo phys.mid phys.hi )
   " assigned-addresses" get-my-property  if   ( phys.hi )
      ." No address property found!" cr
      0 0 rot  exit                            \ Error exit
   then                     ( phys.hi adr len )

   rot >r                   ( adr len )  ( r: phys.hi )
   \ Found assigned-addresses, get address
   begin  dup  while        ( adr len' )  \ Loop over entries
      decode-phys           ( adr len' phys.lo phys.mid phys.hi )
      h# ff and  r@ h# ff and  =  if  ( adr len' phys.lo phys.mid )  \ This one?
         2swap 2drop        ( phys.lo phys.mid )          \ This is the one
         r> exit            ( phys.lo phys.mid phys.hi )
      else                  ( adr len' phys.lo phys.mid ) \ Not this one
         2drop              ( adr len' )
      then                  ( adr len' )
      decode-int drop decode-int drop        \ Discard boring fields
   repeat
   2drop                    ( )
   ." Base address not assigned!" cr

   0 0 r>                   ( 0 0 phys.hi )
;

headers
\ variables/values for sending commands, mapping, etc.
-1 instance value scsi-init-id
```

Code Example 7-3 Sample Driver for Bootable Devices *(Continued)*

```
-1 instance value scsi-chip
h# 20 constant /mbuf
-1 instance value mbuf
-1 instance value mbuf-dma
d# 6 constant /sense
-1 instance value sense-command
-1 instance value sense-cmd-dma
d# 8 constant #sense-bytes
-1 instance value sense-buf
-1 instance value sense-buf-dma
-1 instance value mbuf0
d# 12 constant /cmdbuf
-1 instance value cmdbuf
-1 instance value cmdbuf-dma
-1 instance value scsi-statbuf

\ Mapping and allocation routines for SCSI
: map-scsi-chip  ( -- )
   map-in-broken?  if
      my-space h# 8200.0010 or  get-base-address   ( phys.lo phys.mid phys.hi )
   else
      my-address my-space h# 200.0010 or           ( phys.lo phys.mid phys.hi )
   then                                            ( phys.lo phys.mid phys.hi )

   /scsi map-in  to scsi-chip

   4 dup my-w@ 6 or swap my-w!  \ Enable memory space and bus mastering
   scsi-chip encode-int " address" property
;
: unmap-scsi-chip  ( -- )
   4 dup my-w@ 6 invert and swap my-w!  \ Disable memory space and bus mastering
   scsi-chip /scsi map-out  -1 to scsi-chip
   " address" delete-property
;
: map-scsi  ( -- )
   map-scsi-chip
   \ allocate buffers etc. for "sense" command and status
   ...
;
: unmap-scsi  ( -- )
   \ free buffers etc. for "sense" command and status
   ...
   unmap-scsi-chip
;
\ words related to scsi commands and register access.
...

: reset-my-scsi ( -- error? )   ...  ;
: reset-bus ( -- error? )   ...  ;

: init-n-test  ( -- ok? ) ...  ;
: get-buffers ( -- )  ...  ;
: give-buffers ( -- )  ...  ;
: scsi-selftest ( -- fail? )  ...  ;


d# 512 constant ublock
```

Code Example 7-3 Sample Driver for Bootable Devices *(Continued)*

```
0 instance value /block
0 instance value /tapeblock
instance variable fixed-len?
...

external
: set-timeout  ( n -- ) ...  ;

: send-diagnostic ( -- error? )
   \ run diagnostics and return any error.
   ...
;

: device-present?  ( lun target -- present? ) ...  ;

: mode-sense  ( -- true | block-size false ) ...  ;

: read-capacity  ( -- true | block-size false ) ...  ;

\ Spin up a SCSI disk, coping with a possible wedged SCSI bus
: timed-spin  ( target lun -- ) ...  ;

: disk-r/w-blocks ( addr block# #blocks direction? -- #xfered )
   ...                ( #xfered )
;

\ Execute "mode-sense" command.  If failed, execute read-capacity command.
\ If this also failed, return d# 512 as the block size.
: disk-block-size  ( -- n )
   mode-sense  if  read-capacity  if  d# 512  then  then
   dup to /block
;

: tape-block-size ( -- n ) ...  ;

: fixed-or-variable  ( -- max-block fixed? )  ...  ;

: tape-r/w-some  ( addr block# #blks read? -- actual# error? ) ...  ;

headers

: dma-open  ( -- )  my-address dma-offset 0 d+  my-space /dma map-in  to dma-chip  ;

: dma-close  ( -- )  dma-chip /dma map-out  -1 to dma-chip  ;

\ After any changes to mbuf by CPU or any changes to mbuf-dma by device, synchronize
\ changes by issuing " mbuf mbuf-dma /mbuf dma-sync "
: mbuf-alloc ( -- )
   /mbuf dma-alloc to mbuf
   mbuf /mbuf false dma-map-in  to mbuf-dma
;

\ dma-map-out and dma-free mbuf-dma
: mbuf-free ( -- )
   mbuf mbuf-dma /mbuf dma-sync              \ redundant
   mbuf mbuf-dma /mbuf dma-map-out
   mbuf /mbuf dma-free
```

Code Example 7-3 Sample Driver for Bootable Devices *(Continued)*

```
   -1 to mbuf
   -1 to mbuf-dma
;

external

\ If any routine were using buffers allocated by dma-alloc, and were using DMA mapped
\ by dma-map-in, it would have to dma-sync those buffers after making any changes to
\ them.
: open  ( -- success? )
   dma-open
   " scsi-initiator-id" get-inherited-property 0=  if
      decode-int  to scsi-init-id
      2drop
      map-scsi
      init-n-test                    ( ok? )
      dup if                         ( true )
         get-buffers                 ( true )
      else
         unmap-scsi dma-close        ( false )
      then                           ( success? )
   else
      ." Missing initiator id" cr  false
      dma-close
   then                              ( success? )
;

: close  ( -- )  give-buffers unmap-scsi dma-close  ;

: reset  ( -- )
   dma-open map-scsi
   ...
   reset-my-scsi drop reset-bus drop
   unmap-scsi dma-close
;

\ If scsi-selftest were actually using buffers allocated by mbuf-alloc, it would
\ have to do dma-sync after any changes to mbuf or mbuf-dma.
: selftest  ( -- fail? )
   map-scsi
   mbuf-alloc
   scsi-selftest
   mbuf-free
   unmap-scsi
;

headers

new-device  \ Start of child block device
            \ Missing "reg" property indicates this is a SCSI "wild-card" node
   " sd" device-name
   " block" device-type

   0 instance value offset-low
   0 instance value offset-high
   0 instance value label-package
```

Code Example 7-3 Sample Driver for Bootable Devices *(Continued)*

```
0 instance value deblocker





\ The "disk-label" package interprets any partition information contained in
\ the disk label. The "load" method of the "block" device uses the load method
\ provided by "disk-label"
: init-label-package  ( -- okay? )
   0 to offset-high  0 to offset-low
   my-args  " disk-label"  $open-package to label-package
   label-package  if
      0 0  " offset" label-package $call-method
      to offset-high to offset-low
      true
   else
      ." Can't open disk label package"  cr  false
   then
;

: init-deblocker  ( -- okay? )
   " "  " deblocker"  $open-package  to deblocker
   deblocker  if
      true
   else
      ." Can't open deblocker package"  cr  false
   then
;

: device-present? ( lun target -- present? )
   " device-present?" $call-parent
;

\ The following methods are needed for "block" device:
\ open, close, selftest, reset, read, write, load, seek, block-size,
\ max-transfer,read-blocks, write-blocks.
\ Carefully notice the relationship between the methods for the "block" device
\ and the methods pre-defined for "disk-label" and "deblocker"

external  \ external methods for "block" device ( "sd" node)

: spin-up  ( -- )  my-unit  " timed-spin" $call-parent  ;

: open  ( -- ok? )
   my-unit device-present?  0=  if  false exit  then
   spin-up      \ Start the disk if necessary

   init-deblocker  0=  if  false exit  then
   init-label-package  0=  if
      deblocker close-package false exit
   then
   true
;

: close  ( -- )
   label-package close-package  0 to label-package
```

Code Example 7-3 Sample Driver for Bootable Devices *(Continued)*

```
      deblocker close-package  0 to deblocker
   ;




   : selftest ( -- fail? )
      my-unit device-present?  if
         " send-diagnostic" $call-parent  ( fail? )
      else
         true                             ( error )
      then
   ;
   : reset  ( -- )  ...   ;

   \ The "deblocker" package assists in the implementation of byte-oriented read and
   \ write methods for disks and tapes. The deblocker provides a layer of buffering
   \ to implement a high level byte-oriented interface "on top of" a low-level
   \ block-oriented interface.

   \ The "seek", "read" and "write" methods of this block device use corresponding
   \ methods provided by "deblocker"

   \ In order to be able to use the "deblocker" package this device has to define the
   \ following six methods, which the deblocker uses as its low-level interface
   \ to the device:
   \ 1) block-size, 2) max-transfer, 3) read-blocks, 4) write-blocks 5) dma-alloc and
   \ 6) dma-free

   : block-size ( -- n )   " disk-block-size" $call-parent  ;
   : max-transfer ( -- n ) block-size h# 40 * ;

   : read-blocks  ( addr block# #blocks -- #read )
      true " disk-r/w-blocks" $call-parent
   ;
   : write-blocks  ( addr block# #blocks -- #written )
      false " disk-r/w-blocks" $call-parent
   ;
   : dma-alloc ( #bytes -- vadr ) " dma-alloc" $call-parent  ;
   : dma-free  ( vadr #bytes -- ) " dma-free" $call-parent  ;

   : seek  ( offset.low offset.high -- okay? )
      offset-low offset-high  x+  " seek"   deblocker $call-method
   ;
   : read  ( addr len -- actual-len )  " read"  deblocker $call-method  ;
   : write ( addr len -- actual-len )  " write" deblocker $call-method  ;
   : load  ( addr -- size )            " load"  label-package $call-method  ;

finish-device  \ finishing "block" device "sd"

headers

new-device  \ start of child byte device
            \ missing "reg" indicates this is a SCSI "wild-card" node
   " st" device-name
   " byte" device-type
```

Code Example 7-3 Sample Driver for Bootable Devices *(Continued)*

```
false instance value write-eof-mark?
instance variable file-mark?
true instance value scsi-tape-first-install
: scsi-tape-rewind     ( -- [[xstatbuf] f-hw] error? ) ... ;

: write-eof  ( -- [[xstatbuf] f-hw] error? ) ...  ;
0 instance value deblocker
: init-deblocker  ( -- okay? )
   " "  " deblocker"  $open-package  to deblocker
   deblocker  if
      true
   else
      ." Can't open deblocker package"  cr  false
   then
;

: flush-deblocker  ( -- )
   deblocker close-package  init-deblocker drop
;
: fixed-or-variable ( -- max-block fixed? )
   " fixed-or-variable" $call-parent
;
: device-present? ( lun target -- present? )
   " device-present?" $call-parent
;

\ The following methods are needed for "byte" devices:
\ open, close, selftest, reset, read, write, load, seek,  block-size,
\ max-transfer, read-blocks, write-blocks. Carefully notice the relationship
\ between the methods for "byte" devices and the methods pre-defined for the
\ standard deblocker package.

external  \ external methods for "byte" device ( "st" node)

\ The "deblocker" package assists in the implementation of byte-oriented read
\ and write methods for disks and tapes. The deblocker provides a layer of
\ buffering to implement a high level byte-oriented interface "on top of" a
\ low-level block-oriented interface.

\ The "read" and "write" methods of this "byte" device use the corresponding
\ methods provided by the "deblocker"

\ In order to be able to use the "deblocker" package this device has to define the
\ following six methods which the deblocker uses as its low-level interface to
\ the device: 1) block-size, 2) max-transfer, 3) read-blocks, 4) write-blocks
\ 5) dma-alloc and 6) dma-free

: block-size  ( -- n )   " tape-block-size" $call-parent  ;

: max-transfer  ( -- n )
   fixed-or-variable  ( max-block fixed? )
   if
     h# fe00  over  / *  \ Use the largest multiple of /tapeblock that is <= h# fe00
   then
;

: read-blocks  ( addr block# #blocks -- #read )
```

Code Example 7-3 Sample Driver for Bootable Devices *(Continued)*

```
      file-mark? @  0=  if
         true " tape-r/w-some" $call-parent  file-mark? !   ( #read )
      else
         3drop 0
      then
   ;
   : write-blocks  ( addr block# #blocks -- #written )
      false " tape-r/w-some" $call-parent file-mark? !
   ;

   : dma-alloc  ( #bytes -- vaddr )  " dma-alloc" $call-parent  ;

   : dma-free   ( vaddr #bytes -- )  " dma-free" $call-parent  ;

   : open  ( -- okay? )  \ open for tape
      my-unit  device-present?  0=  if  false exit  then
      scsi-tape-first-install  if
         scsi-tape-rewind  if
            ." Can't rewind tape" cr
            0= if  drop  then
            false exit
         then
         false to scsi-tape-first-install
      then
      \ Set fixed-len? and /tapeblock
      fixed-or-variable 2drop
      init-deblocker  0=  if  false exit  then
      true
   ;

   : close  ( -- )
      deblocker close-package  0 to deblocker
      write-eof-mark?  if
         write-eof  if
            ." Can't write EOF Marker."
            0=  if  drop  then
         then
      then
   ;

   : reset  ( -- )  ...   ;

   : selftest ( -- fail? )
      my-unit device-present?  if
         " send-diagnostic" $call-parent  ( fail? )
      else
         true                             ( error )
      then
   ;

   : read  ( addr len -- actual-len )  " read"  deblocker $call-method  ;

   : write ( addr len -- actual-len )
      true to write-eof-mark?
      " write" deblocker $call-method
   ;
```

Code Example 7-3 Sample Driver for Bootable Devices *(Continued)*

```
   : load  ( addr -- size )
      \ use my-args to get tape file-no
      ...  ( addr file# )

      \ position at requested file
      ...
      dup  begin                  ( start-addr next-addr )
         dup max-transfer read    ( start-addr next-addr #read )
         dup 0>                   ( start-addr next-addr #read got-some? )
      while                       ( start-addr next-addr #read )
         +                        ( start-addr next-addr' )
      repeat                      ( start-addr end-addr 0 )
      drop swap -                 ( size )
   ;

   : seek  ( byte# file# -- error? )
      \ position at requested file
      ...                         ( byte# )

      flush-deblocker             ( byte# )
      begin  dup 0>  while        ( #remaining )
         " mbuf0" $call-parent
         over ublock min  read    ( #remaining #read )
         dup  0=  if              ( #remaining 0 )
            2drop  true
            exit                  ( error )
         then                     ( #remaining #read )
         -                        ( #remaining' )
      repeat                      ( 0 )
      drop false                  ( no-error )
   ;

finish-device  \ finishing "byte" device "st"

fcode-end
```

8 Network Devices

Network devices are packet-oriented devices capable of sending and receiving packets addressed according to IEEE 802.3 (Ethernet). Open Firmware firmware typically uses network devices for diskless booting. The standard `obp-tftp` support package assists in the implementation of the `load` method for this device type.

This chapter describes how to implement network device drivers. The example is based upon a hypothetical network device that is similar to existing devices. This hypothetical device was used to reduce the number of details that might otherwise obscure the design of the driver. A driver for a real-world device is likely to be more complex in that state of the art Ethernet chip sets tend to have somewhat more elaborate schemes for managing transmit and receive buffers.

Required Methods

The network device FCode must set the value of its `device_type` property to `network` and must implement the following methods:

open (-- ok?)

Prepare the device for use by performing those hardware-dependent actions required to allocate resources and start the device. Create a `"mac-address"` property with the value returned by `mac-address`. Return `true` if the `open` method succeeds; `false` otherwise.

close (--)

Return the device to its “not-in-use” state by performing those hardware-dependent actions required to stop the device and de-allocate resources.

read (addr len -- actual)

Receive a network packet, placing at most the first *len* bytes in memory at *addr*. Return the *actual* number of bytes received (not the number copied), or -2 if no packet is currently available. Packets with hardware-detected errors are discarded as though they were not received. Do not wait for a packet (non-blocking).

The received packet format is shown in Figure 8.

write (addr len -- actual)

Transmit the network packet of size *len* bytes starting at memory address *addr*. The format of the buffer at *addr* is shown in Figure 8. Return the number of bytes actually transmitted. The packet must be complete with all addressing information, including source hardware address, as shown in Figure 9.

load (addr -- len)

Read the default stand-alone program into memory starting at *addr* using the default network booting protocol. *len* is the size in bytes of the program read in.

A suitable definition of `load` is:

Code Example 8-1

```
: load ( addr -- len )
   my-args " obp-tftp" $open-package >r  ( addr ) ( r: ihandle )
   r@ 0= abort" Can't open TFTP package" ( addr ) ( r: ihandle )
   >r " load" r@ $call-method            ( len )  ( r: ihandle )
   r> close-package
;
```

Required Device Properties

The required properties for a `network` device are:

Table 26 Required Network Device Properties

Name	Typical Value
`name`	`" INTL,my-net"`
`reg`	list of registers *{device-dependent}*
`device_type`	`" network"`
`mac-address`	8 0 0x20 0x0c 0xea 0x41 *{the MAC address currently being used.}*

Optional Device Properties

Several other properties may be declared for `network` devices:

Table 27 Optional Network Device Properties

Property Name	Typical Property Value
`max-frame-size`	0x4000
`address-bits`	48
`local-mac-address`	8 0 0x20 0x0c 0xea 0x41 *{the built-in Media Access Control address.}*

`network` Device Driver Issues

`write` Buffer Format

The `write` method of a `network` device driver receives a buffer whose contents are shown in Figure 8. It should be noted that the driver is not intended to interpret the “length / Ethernet type” field since it may contain either a length or an Ethernet type. It is the responsibility of whoever is calling the `write` method to fill in that field appropriately.

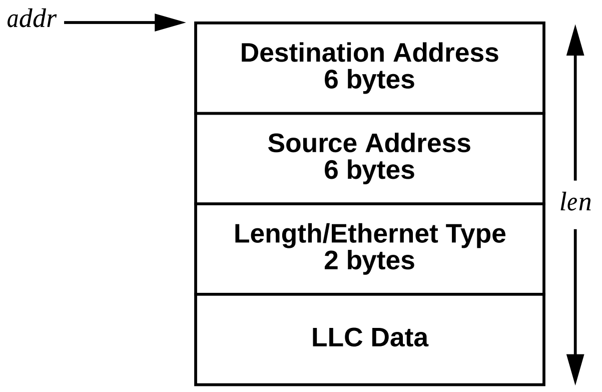

Figure 8 `write` Method Input Buffer Format

The driver is responsible for ensuring that the packet that is sent on the network has the form shown in Figure 9. In reality, the hardware will almost certainly automatically supply the “preamble”, “start frame delimiter” and “frame check sequence”. Hardware will often provide the “pad” in those cases where the “data” is shorter than the minimum required 64 bytes. However, the driver must supply any of this information that the hardware does not.

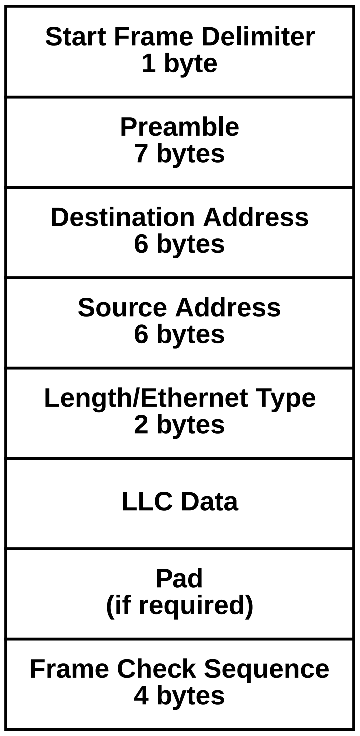

Figure 9 Network Packet Format

`read` Buffer Format

Because of the ambiguity of the “length/Ethernet type” field as shown in Figure 8, a `network` driver is not expected to and should not try to remove any “pad” bytes that may be passed to it by the hardware. The driver should simply pass the data supplied to it by the hardware (subject to the limitations of its *len* argument).

Use of DMA

The `obp-tftp` package is not required by *IEEE Standard 1275-1994* to provide packets in buffers that are suitable for DMA. To use DMA, a `network` driver must:

- Provide its own DMA-accessible packet buffers with `dma-alloc` and `dma-map-in`.
- Flush any caches with `dma-sync` and copy received data from a DMA buffer into a buffer provided as an argument to the `read` method.
- Copy data to be transmitted from a buffer supplied as an argument to the `write` method into a DMA buffer and flush any caches with `dma-sync`.
- Return its DMA buffers with `dma-free` and `dma-map-out`.

`selftest`

Note – United States Patent No. 4,633,466, "Self Testing Data Processing System with Processor Independent Test Program", issued December 30, 1986 may apply to some or all elements of Open Firmware selftest. Anyone implementing Open Firmware should take such steps as may be necessary to avoid infringement of that patent and any other applicable intellectual property rights.

The example below shows a bootable network driver. It implements the `selftest` method callable by Open Firmware `test` and `test-all` commands and the `watch-net` method callable by Open Firmware `watch-net` and `watch-net-all` commands.

Since the inclusion of a `selftest` method on a plug-in card may infringe the patent mentioned above, writers of drivers for network plug-in cards may wish to omit the `selftest` method. However, writers of drivers for network devices that are built onto a system motherboard are encouraged to include the `selftest` method.

Device Driver Examples

Simple Bootable Network Device Example

The example below shows a complete version of a simple bootable network driver.

Code Example 8-2 Simple Bootable Ethernet Driver

```
\ This driver assumes a hypothetical Ethernet adapter as described below.
\ While it would be possible to design an Ethernet adapter similar to this
\ (and, in fact, many early Ethernet adapters were reminiscent of this design),
\ in practice modern Ethernet adapters are somewhat more complicated.
\
\ This hypothetical adapter is deficient in at least the following ways:
\ a) The need to copy packets in and out through a single byte-wide
```

Code Example 8-2 Simple Bootable Ethernet Driver *(Continued)*

```
\    register is a performance bottleneck.  Most modern Ethernet adapters use DMA.
\ b) The single transmit buffer prevents the adapter from sending
\    consecutive packets with the mininum Ethernet interpacket gap.
\ c) There is no provision for interrupts.  This does not affect FCode
\    drivers, which assume a single-task polled environment, but it would
\    be a problem for a real system.
\
\ The hypothetical adapter has a control register with six bits:
\ 01   \ Reset chip
\ 02   \ Enable reception
\ 04   \ Release receive buffer
\ 08   \ Start transmission
\ 10   \ Enable promiscuous reception
\ 20   \ Enable internal loopback
\ Writing a one to a control register bit causes it to perform the indicated action.

\ There is a single transmit buffer.  When the xmit-status register is not zero, the
\ hardware is ready to send a packet.  To send the packet, the host gives the packet
\ to the adapter by writing it to the xmit-fifo register one byte at a time.  Then
\ the host writes the number of bytes to transmit to the xmit-len register.  Finally,
\ the host writes "8" to the control register to begin the transmission.  The adapter
\ responds by setting the xmit-status register to zero and initiating transmission.
\ When transmission is complete, the adapter sets the xmit-status register to one if
\ the transmission was successful, or to some value other than zero or one to indicate
\ an error.  If the value written to the xmit-len register is greater than the number
\ of bytes that were written to the xmit-fifo register, the adapter transmits zeroes
\ after transmitting the bytes that were written to the FIFO.

\ There are numerous receive buffers organized as a queue.  When a packet is received,
\ the adapter sets the rcv-rdy register to the number of currently-available packets
\ and sets the rcv-len register to the length of the first available packet.  If that
\ packet is defective, the adapter also sets the 0x8000 bit of the rcv-len register.
\ The host accepts the packet by copying one byte at a time from the rcv-fifo register.
\ When the host has copied all the bytes of that packet that it wants, the host writes
\ "4" to the control register, which causes the adapter to make that packet buffer
\ available for other incoming packets, to decrement the rcv-rdy buffer available for
\ other incoming packets, to decrement the rcv-rdy register, and to update the rcv-len
\ register to reflect the next available packet.

\ The adapter reports the 6-byte Ethernet address that its manufacturer assigned to it
\ via six registers beginning with the "local-addr" register. The adapter compares
\ incoming packets to the 6 registers beginning with the "unicast-addr" register,
\ receiving those whose destination address matches and discarding others (except for
\ broadcast packets, which it always receives).  The host must set this unicast
\ address before enabling reception.  The host software decides whether to use the
\ manufacturer-assigned Ethernet address or some other Ethernet address for this
\ purpose.

fcode-version2
hex
headers
: copyright  ( -- )
   ." Copyright (c) 1995-1996 FirmWorks.  All Rights Reserved." cr
;
\ Register offsets from the adapter’s base address

0 constant control      \ 1 byte W/O - writing one bits causes things to happen
```

Code Example 8-2 Simple Bootable Ethernet Driver *(Continued)*

```
2 constant unicast-addr \ 6 bytes R/W - Ethernet address for reception

8 constant xmit-status  \ 1 byte - 0 => busy  1 => okay  else => error
9 constant xmit-fifo    \ 1 byte - write repetitively to setup packet
a constant xmit-len     \ 16 bits - length of packet to send

c constant rcv-rdy      \ 1 byte - count of waiting packets
d constant rcv-fifo     \ 1 byte - read repetitively to remove first packet
e constant rcv-len      \ 16 bits

10 constant local-addr  \ 6 bytes R/O - Factory-assigned Ethernet address

16 constant /regs       \ Total size of adapter’s register bank

: map-in   ( addr space size -- virt )  " map-in"  $call-parent  ;
: map-out  ( virt size -- )             " map-out" $call-parent  ;

: my-w@ ( offset -- w ) my-space + " config-w@" $call-parent ;
: my-w! ( w offset -- ) my-space + " config-w!" $call-parent ;

" FirmWorks,ethernet" device-name
" network" device-type
" FirmWorks,54321" model

\ Some of Apple’s Open Firmware implementations have a bug in their map-in method. The
\ bug causes phys.lo and phys.mid to be treated as absolute addresses rather than
\ offsets even when working with relocatable addresses.

\ To overcome this bug, the Open Firmware Working Group in conjunction with Apple has
\ adopted a workaround that is keyed to the presence or absence of the add-range method
\ in the PCI node. If the add-range method is present in an Apple ROM, the map-in
\ method is broken. If the add-range property is absent, the map-in method behaves
\ correctly.

\ The following methods allow the FCode driver to accomodate both broken and working
\ map-in methods.

: map-in-broken?  ( -- flag )
   \ Look for the method that is present when the bug is present
   " add-range"  my-parent  ihandle>phandle   ( adr len phandle )
   find-method  dup  if  nip  then            ( flag )  \ Discard xt if present
;

\ Return phys.lo and phys.mid of the address assigned to the PCI base address
\ register indicated by phys.hi .
: get-base-address  ( phys.hi -- phys.lo phys.mid phys.hi )
   " assigned-addresses" get-my-property  if   ( phys.hi )
      ." No address property found!" cr
      0 0 rot  exit                           \ Error exit
   then                    ( phys.hi adr len )

   rot >r                  ( adr len )  ( r: phys.hi )
   \ Found assigned-addresses, get address
   begin  dup  while       ( adr len' )  \ Loop over entries
      decode-phys          ( adr len' phys.lo phys.mid phys.hi )
      h# ff and  r@ h# ff and  =  if  ( adr len' phys.lo phys.mid )  \ This one?
         2swap 2drop       ( phys.lo phys.mid )          \ This is the one
         r> exit             ( phys.lo phys.mid phys.hi )
      else                   ( adr len' phys.lo phys.mid ) \ Not this one
         2drop               ( adr len' )
      then                   ( adr len' )
      decode-int drop decode-int drop        \ Discard boring fields
   repeat
   2drop                     ( )
   ." Base address not assigned!" cr

   0 0 r>                    ( 0 0 phys.hi )
;

\ String comparision
: $=  ( adr0 len0 adr1 len1 -- equal? )
   2 pick <>  if  3drop false exit  then  ( adr0 len0 adr1 )
   swap comp 0=
;





\ Define "reg" property
\ PCI Configuration Space
my-address my-space encode-phys  0 encode-int encode+  0 encode-int encode+

\ Memory Space Base Address Register 10
my-address my-space 0200.0010 or encode-phys  encode+
0 encode-int encode+  /regs encode-int encode+

\ PCI Expansion ROM
my-address my-space h# 200.0030 or encode-phys encode+
0 encode-int encode+ h# 10.0000 encode-int encode+
" reg" property

-1 instance value chipbase

: map-regs  ( -- )
   map-in-broken?  if
      my-space h# 8200.0010 or  get-base-address   ( phys.lo phys.mid phys.hi )
   else
      my-address my-space h# 200.0010 or           ( phys.lo phys.mid phys.hi )
   then                                            ( phys.lo phys.mid phys.hi )

   /regs map-in  to chipbase

   4 dup my-w@ 4 or swap my-w!  \ Enable memory space
   chipbase encode-int " address" property
;
: unmap-regs  ( -- )
   4 dup my-w@ 4 invert and swap my-w!  \ Disable memory space
   chipbase /regs map-out  -1 to chipbase
   " address" delete-property
;
: e@   ( register -- byte )     chipbase + rb@  ;
: e!   ( byte register -- )     chipbase + rb!  ;
: ew@  ( register -- 16-bits )  chipbase + rw@  ;
```

Code Example 8-2 Simple Bootable Ethernet Driver *(Continued)*

```
: ew!  ( 16-bits register -- )  chipbase + rw!  ;

: control-on  ( control-bit -- )  control e@ or control e!  ;
: control-off  ( control-bit -- )  invert control e@ and control e!  ;
: reset-chip  ( -- )  1 control e! ;
: receive-on  ( -- )  2 control-on  ;
: return-buffer  ( -- )  4 control-on  ;
: start-xmit  ( -- )  8 control-on ;
: promiscuous  ( -- )  10 control-on  ;
: loopback-on  ( -- )  20 control-on  ;
: loopback-off  ( -- )  20 control-off  ;

: receive-ready?  ( -- #pkts-waiting )  rcv-rdy e@  ;
: wait-for-packet  ( -- )  begin  key?  receive-ready?  or  until  ;

\ Create local-mac-address property from the information in the chip
map-regs
6 alloc-mem                                             ( mem-addr )
6 0  do  local-addr i + rb@  over i + c!  loop          ( mem-addr )
6 2dup  encode-string  " local-mac-address" property    ( mem-addr 6 )
free-mem
unmap-regs
: initchip  ( -- )
   reset-chip

   \ Ask the host system for the station address and give it to the adapter
   mac-address  0  do                   ( addr )
      dup  i + c@   unicast-addr i + e!  ( addr )
   loop   drop

   receive-on  \ Enable reception
;
: net-init  ( -- succeeded? )
   loopback-on  loopback-test  loopback-off  if  init-chip true  else  false  then
;
\ Check for incoming Ethernet packets while using promiscuous mode.
: watch-test  ( -- )
   ." Looking for Ethernet packets." cr
   ." '.' is a good packet.  'X' is a bad packet." cr
   ." Press any key to stop." cr
   Begin
      wait-for-packet
      receive-ready?  if
         rcv-len ew@ 8000 and 0=  if  ." ."  else  ." X"  then
         return-buffer
      then
      key? dup  if  key drop  then
   until
;
: (watch-net)  ( -- )
   map-regs
   promiscuous
   net-init  if  watch-test reset-chip  then
   unmap-regs
;
: le-selftest  ( -- passed? )
   net-init
   dup  if  net-off  then
;

external
: read  ( addr requested-len -- actual-len  )
   \ Exit if packet not yet available
   receive-ready? 0=  if  2drop -2 exit  then

   rcv-len ew@  dup 8000 and =  if    ( addr requested-len packet-len )
      3drop  return-buffer   \ Discard bad packet
      -1 exit
   then                                   ( addr requested-len packet-len )

   \ Truncate to fit into the supplied buffer
   min                                    ( addr actual-len )

   \ Note: For a DMA-based adapter, the driver would have to synchronize caches (e.g.
   \ with "dma-sync") and copy the packet from the DMA buffer into the result buffer.

   tuck  bounds  ?do  mem-port i c!  loop   ( actual-len )
   return-buffer                            ( actual-len )
;
: close  ( -- )  reset-chip  unmap-regs  ;

: open  ( -- ok? )
   map-regs
   mac-address  encode-string   " mac-address" property
   initchip
   my-args  " promiscuous" $=  if  promiscuous  then

   \ Note: For a DMA-based adapter, the driver would have to allocate DMA memory for
   \ packet buffers, construct buffer descriptor data structures, and possibly
   \ synchronize caches (e.g. with "dma-sync").

   true
;
: write  ( addr len -- actual )
   begin  xmit-status e@  0<>  until

   \ Note: For a DMA-based adapter, the driver would have to copy the
   \ packet into the DMA buffer and synchronize caches (e.g. with "dma-sync").

   \ Copy packet into chip
   tuck bounds  ?do  i c@   xmit-fifo e!  loop

   \ Set length register
   dup  h# 64 max  xmit-len ew!

   start-xmit
;

: load  ( addr -- len )
   " obp-tftp" find-package  if      ( addr phandle )
      my-args rot  open-package      ( addr ihandle|0 )
   else                              ( addr )
      0                              ( addr 0 )
   then                              ( addr ihandle|0 )
```

Code Example 8-2 Simple Bootable Ethernet Driver *(Continued)*

```
   dup  0=  abort" Can’t open obp-tftp support package"  ( addr ihandle )

   >r
   " load" r@ $call-method           ( len )
   r> close-package
;

: selftest  ( -- failed? )
   map-regs
   le-selftest 0=
   unmap-regs
;

: watch-net  ( -- )
   selftest 0=  if  (watch-net)  then
;
fcode-end
```

9 Serial Devices

Serial devices are byte-oriented, sequentially-accessed devices such as asynchronous communication lines (often attached to a “dumb” terminal).

Required Methods

The `serial` device driver must declare the `serial` device type, and must implement the methods `open` and `close`, as well as the following:

install-abort (--)
Instruct the driver to begin periodic polling for a keyboard abort sequence. `install-abort` is executed when the device is selected as the console input device.

read (addr len -- actual)
Read *len* bytes of data from the device into memory starting at *addr*. Return the number of bytes actually read, *actual*, or -2 if no bytes are currently available from the device. -1 is returned if other errors occur.

remove-abort (--)
Instruct the driver to cease periodic polling for a keyboard abort sequence. `remove-abort` is executed when the console input device is changed from this device to another.

write (addr len -- actual)
Write *len* bytes of data to the device from memory starting at *addr*. Return the number of bytes actually written, *actual*.

Required Properties

The standard properties of a serial driver are:

Table 28 Serial Driver Required Properties

Property Name	Value
`name`	`" INTL,thingy"`
`reg`	*list of registers { device-dependent}*
`device_type`	`" serial"`

Device Driver Examples

The examples that follow are serial device drivers for the Zilog 8530 SCC (UART) chip.

- The first sample is a short driver which simply creates a device node and declare the properties for the device.
- The second sample shows the complete serial device driver. The `open` method accepts an argument of the form [*p*][,*s*] where:
  - *p* is an optional argument indicating which port of the device is to be used. Valid values are `a` and `b`. If *p* is not specified, Port A is used.
  - *s* is an optional argument specifying the speed to which the port should be set in decimal. Valid values are 4800, 9600, 19200 and 38400. If *s* is not specified, 9600 is used.

Simple Serial FCode Program

Code Example 9-1 Simple Serial Device Driver

```
\ This driver creates a device node and publishes the minimum required set of
\ properties.
fcode-version2

hex
" INTL,zs" device-name
" serial"  device-type

\ Define "reg" property
\ PCI Configuration Space
my-address my-space encode-phys  0 encode-int encode+  0 encode-int encode+

\ Memory Aperture
my-address my-space 0200.0010 or encode-phys  encode+
0 encode-int encode+  8 encode-int encode+

\ PCI Expansion ROM
my-address my-space h# 200.0030 or encode-phys encode+
0 encode-int encode+ h# 10.0000 encode-int encode+
"reg" property

fcode-end
```

Complete Serial FCode Program

Code Example 9-2 Complete Serial FCode Program

```
\ Complete Serial driver.
\ In addition to publishing properties, this sample driver provides methods to
\ initialize, test, access and control the serial ports.
\
\ The following main methods are provided:
\ - usea  ( -- )
\   Selects serial port A. All subsequent operations will be directed to port A.
\ - useb  ( -- )
\   Selects serial port B. All subsequent operations will be directed to port B.
\ - uemit  ( char -- )
\   Emits a given character to the selected serial port.
```

Code Example 9-2 Complete Serial FCode Program *(Continued)*

```
\ - ukey  ( -- char )
\   Retrieves a character from the selected serial port.
\ - read  ( addr len -- #read )
\  Reads "len" number of characters from the selected port, and stores them at "addr".
\   Returns the actual number read.
\
\
\ - write  ( addr len -- #written )
\  Writes "len" number of characters from the buffer located at "addr" to the selected
\   port. Returns the actual number written.
\ - inituarts  ( -- )
\   Initializes both serial ports A and B.
\ - open  ( -- okay? )
\   Maps in the uart chip. Selects port A on default, then checks my-args. If port B
\   was specified, then selects port B instead.
\ - close  ( -- )
\   Unmap the uart chip.
\ - selftest  ( -- )
\   Performs selftest on both Port A and B.
\ - install-abort  ( -- )
\   Sets up alarm to do poll-tty every 10 milliseconds.
\ - remove-abort  ( -- )
\   Removes the poll-tty alarm.

fcode-version2
hex
headers

: copyright  ( -- )
   ." Copyright (c) 1995-1996 FirmWorks.  All Rights Reserved." cr
;

" INTL,zs" device-name
" serial" device-type

8 constant /regs       \ Total size of adapter's register bank

\ Define "reg" property
\ PCI Configuration Space
my-address my-space encode-phys  0 encode-int encode+  0 encode-int encode+

\ Memory Aperture
my-address my-space 0200.0010 or encode-phys  encode+
0 encode-int encode+  /regs encode-int encode+

\ PCI Expansion ROM
my-address my-space h# 0200.0030 or encode-phys encode+
0 encode-int encode+ h# 10.0000 encode-int encode+

" reg" property

: map-in  ( phys.lo phys.mid phys.hi size -- virt )  " map-in" $call-parent  ;
: map-out ( virt size -- )  " map-out" $call-parent  ;

: my-w@ ( offset -- w ) my-space + " config-w@" $call-parent ;
: my-w! ( w offset -- ) my-space + " config-w!" $call-parent ;
```

Code Example 9-2 Complete Serial FCode Program *(Continued)*

```
: /string  ( addr len n -- addr+n len-n )  tuck  -  -rot  +  swap  ;

headerless

\ Some of Apple’s Open Firmware implementations have a bug in their map-in method. The
\ bug causes phys.lo and phys.mid to be treated as absolute addresses rather than
\ offsets even when working with relocatable addresses.
\ To overcome this bug, the Open Firmware Working Group in conjunction with Apple has
\ adopted a workaround that is keyed to the presence or absence of the add-range method
\ in the PCI node. If the add-range method is present in an Apple ROM, the map-in
\ method is broken. If the add-range property is absent, the map-in method behaves
\ correctly.
\ The following methods allow the FCode driver to accomodate both broken and working
\ map-in methods.

: map-in-broken?  ( -- flag )
   \ Look for the method that is present when the bug is present
   " add-range"  my-parent  ihandle>phandle   ( adr len phandle )
   find-method  dup  if  nip  then            ( flag )  \ Discard xt if present
;
\ Return phys.lo and phys.mid of the address assigned to the PCI base address
\ register indicated by phys.hi .
: get-base-address  ( phys.hi -- phys.lo phys.mid phys.hi )
   " assigned-addresses" get-my-property  if   ( phys.hi )
      ." No address property found!" cr
      0 0 rot  exit                            \ Error exit
   then                   ( phys.hi adr len )

   rot >r                 ( adr len )  ( r: phys.hi )
   \ Found assigned-addresses, get address
   begin  dup  while      ( adr len' )  \ Loop over entries
      decode-phys         ( adr len' phys.lo phys.mid phys.hi )
      h# ff and  r@ h# ff and  =  if  ( adr len' phys.lo phys.mid )  \ This one?
         2swap 2drop      ( phys.lo phys.mid )          \ This is the one
         r> exit          ( phys.lo phys.mid phys.hi )
      else                ( adr len' phys.lo phys.mid ) \ Not this one
         2drop            ( adr len' )
      then                ( adr len' )
      decode-int drop decode-int drop        \ Discard boring fields
   repeat
   2drop                  ( )
   ." Base address not assigned!" cr

   0 0 r>                 ( 0 0 phys.hi )
;

headers

   -1 instance value  uartbase
    0 instance value  uart        \ define uart as an "per-instance" value.
h# ff instance value  mask-#data  \ mask for #data bits
    6 instance value  tc         \ Baud rate time constant. Init’d to value for 9600
 true instance value  usea?       \ Which port did user specify?
```

Code Example 9-2 Complete Serial FCode Program *(Continued)*

```
\ The following line assumes that A2 selects the channel within the chip
: usea   ( -- )    uartbase 4 + to uart  ;
: useb   ( -- )    uartbase to uart  ;
: uctl!  ( c -- )  uart  rb!  ;
: uctl@  ( -- c )  uart  rb@  ;

\ The following line assumes that A1 chooses the command vs. data port
: udata!  ( c -- )  uart  2 + rb!  ;
: udata@  ( -- c )  uart  2 + rb@  ;

\ Write the initialization sequence to both UARTs
: inituart  ( -- )
\ Reg      Value          Description
  9 uctl!   2 uctl!     \ Don't respond to intack cycles (02)
  4 uctl!  44 uctl!     \ No parity (00), 1 stop bit (04), x16 clock (40)
  3 uctl!  c0 uctl!     \ receive 8 bit characters (c0)
  5 uctl!  60 uctl!     \ transmit 8 bits (60)
  e uctl!  82 uctl!     \ Processor clock is baud rate source (02)
  b uctl!  55 uctl!     \ TRxC = xmit clk (01), enable TRxC (04), Tx clk is baud (10),
                        \ Rx clk is baud (40)
  c uctl!  tc uctl!     \ Time constant low
  d uctl!   0 uctl!     \ Time constant high
  3 uctl!  c1 uctl!     \ receive 8 bit characters (c0), enable (01)
  5 uctl!  68 uctl!     \ transmit 8 bits (60), enable (08)
  e uctl!  83 uctl!     \ Processor clock is baud rate source (02), Tx enable (01)
  0 uctl!  10 uctl!     \ Reset status bit latches
;

: inituarts  ( -- )  usea inituart  useb inituart  ;

inituarts

\ Test for "break" character received.
: ubreak?  ( -- break? )  10 uctl!  uctl@  h# 80 and  0<>  ;

: clear-break  ( -- )  \ Clear the break flag
   begin  ubreak? 0=  until  \ Let break finish
   udata@ drop               \ Eat the null character
   30 uctl!                  \ Reset errors
;

1 constant RXREADY           \ received character available
4 constant TXREADY           \ transmit buffer empty

: uemit? ( -- emit? ) uctl@ TXREADY and 0<>  ;
: uemit ( char -- )  begin  uemit?  until  udata!  ;

: ukey? ( -- key? )  uctl@ RXREADY and 0<>  ;
: ukey  ( -- key )  begin  ukey?  until  udata@  ;

: poll-tty  ( -- )
   ubreak?  if  clear-break  user-abort  then
;
```

Code Example 9-2 Complete Serial FCode Program *(Continued)*

```
: which-port?  ( arg-str arg-len -- speed-str speed-len )
   ascii ,  left-parse-string  if   ( speed-str speed-len port-str )
      c@ lcc  case                  ( speed-str speed-len       )
         ascii a of  true   endof   ( speed-str speed-len flag  )
         ascii b of  false  endof   ( speed-str speed-len       )
         1 throw  \ Throw an error on an unrecognized port letter
      endcase                       ( speed-str speed-len flag  )
   else                             ( speed-str speed-len port-str )
      drop true                     ( speed-str speed-len flag  )
   then                             ( speed-str speed-len flag  )
   to usea?                         ( speed-str speed-len       )
;
: set-baud-rate  ( speed-str speed-len -- )
   dup if
      base @ decimal  ( base )  \ Change to decimal at run-time and save old base
      $number  if     ( base | base baud-rate )
         base !  2 throw  \ Throw an error on a non-decimal speed specification
      then            ( base baud-rate )
      case            ( base baud-rate )
         d#  9600 of  6  endof
         d# 38400 of  0  endof
         d# 19200 of  2  endof
         d#  4800 of  e  endof
         drop base !  3 throw  \ Throw an error on an invalid speed specification
      endcase         ( base time-constant  )
      swap base !     ( time-constant       )
   else               ( speed-str 0         )
      2drop 6         ( 6                   )  \ Time constant for 9600 (default)
   then               ( time-constant       )
   to tc              (    )
;

external

: open  ( -- okay? )
   map-in-broken?  if
      my-space h# 8200.0010 or  get-base-address   ( phys.lo phys.mid phys.hi )
   else
      my-address my-space h# 200.0010 or           ( phys.lo phys.mid phys.hi )
   then                                            ( phys.lo phys.mid phys.hi )

   /regs map-in  to uartbase

   4 dup my-w@ 4 or swap my-w!  \ Enable memory space
   uartbase encode-int " address" property

   my-args                        ( arg-str arg-len )
   ['] which-port? catch if       ( arg-str arg-len | speed-str speed-len )
      2drop false                 ( false )
   else                           ( speed-str speed-len )
      ['] set-baud-rate catch if  ( speed-str speed-len | <nothing> )
         2drop false              ( false )
      else                        (       )
         usea? if  usea  else  useb  then  inituart  true  ( true )
      then                        ( okay? )
   then                           ( okay? )
;
```

Code Example 9-2 Complete Serial FCode Program *(Continued)*

```
: close  ( -- )
   4 dup my-w@ 4 invert and swap my-w!  \ Disable memory space
   uartbase /regs map-out  -1 to uartbase
   " address" delete-property
;

headers

: utest  ( -- 0 )  h# 7f  bl  do  i uemit  loop 0  ;

external

: selftest  ( -- error? )
   open  0=  if  ." Can't open device" true exit  then
   my-args  if       ( addr )
      c@ lcc  case
         ascii a  of  usea  endof
         ascii b  of  useb  endof
         ( default ) ." Bad zs port letter = " emit close false exit
      endcase
      utest          ( fail? )
   else  \ No port letter so test both ports.
      drop
      usea utest
      useb utest
      or             ( fail? )
   then
   close
;

: read  ( addr len -- #read )  \ #read = -2 == none available right now
   ukey? 0=  if  2drop -2  exit  then      ( addr len )
   tuck                                    ( len addr len )
   begin  dup 0<>   ukey?  and  while      ( len addr len )
      over  ukey mask-#data and swap c!    ( len addr len )
      1 /string                            ( len addr' len' )
   repeat                                  ( len addr' len' )
   nip -                                   ( #read )
;

: write  ( addr len -- #written )
   tuck  bounds ?do   ( len )
      i c@  uemit     ( len )
   loop               ( len )
;

: install-abort  ( -- )  ['] poll-tty d# 10 alarm  ;
: remove-abort   ( -- )  ['] poll-tty 0 alarm  ;

\ "seek" might be implemented to select a load file name
\ Implement "load" ( optional )

fcode-end
```

10 Display Devices

The `display` device type applies to framebuffers and other devices that appear to be memory to the processor with associated hardware to convert the memory image to a visual display. Display devices can be used as console output devices.

Required Methods

To be usable as the console output device during Open Firmware start up, a display device's FCode must set the value of its `device_type` property to `display`. It must also implement the methods `open` and `close`, and may optionally implement the `selftest` method. However, for historical reasons, the `open`, `close` and `selftest` methods are not created directly in the driver FCode as they are for other device types.

Unlike other device types that obtain support services from the system firmware through the `/packages` node, display devices interact with system firmware with a special `defer` word interface used exclusively by display devices.

When writing an FCode program for a display device, you create methods whose behavior is later "installed" into the `defer` words of the display device interface. The FCodes `is-install`, `is-remove` and `is-selftest` are used to attach the methods defined in your FCode program to the `defer` word interface, and to create the `open`, `close` and optional `selftest` routines for your `display` device. In addition, the `set-font` FCode initializes the values of `fontbytes`, `char-height` and `char-width`, all of which are used to configure the low level display device interface `defer` words.

The lists of FCodes specifically designed for use with display devices are listed in Note – Table 82 through Table 89 in Appendix A, "FCode Reference".

Required Properties

Table 29 Required Display Device Properties

Property Name	Typical Value
`name`	`" FirmWorks,generic-vga"`
`reg`	list of registers *{device dependent}*
`device_type`	`" display"` *{required for display devices}*
`character-set`	`" ISO8859-1"` *(device dependent)*
`height`	`#scanlines` *(device dependent)*
`width`	`/scanline` *(device dependent)*
`depth`	`8` *(device dependent)*
`linebytes`	`#scanlines` *(device dependent)*

Structure of a `display` Device Driver

The Open Firmware system firmware provides support to display device drivers with the terminal emulator and the low level display device interface `defer` words. This support requires that an FCode program create certain properties and do certain operations in a certain order at probe time. In addition, the words installed with `is-install` and `is-remove` must perform certain operations in a certain order.

Probe Time Actions

- Create required properties.
- Create manufacturer-specific properties (if any).
- Create terminal emulator properties.
  - `height`, `width`, `depth` and `linebytes`
  - If appropriate, `iso6429-1983-colors`
- `is-install`
- `is-remove`
- `is-selftest`, if desired.

`is-install` Actions

The word whose execution token is installed with `is-install` must:

- Map in the frame buffer. Enable PCI memory and/or I/O space access as required.
- Initialize the graphics hardware.
- Initialize the color palette.
  - `default-colors`
  - `set-colors`
- Initialize `frame-buffer-adr`
- Create the `address` property.
- Install the `fb8` package.
  - `default-font`
  - `set-font`
  - `fb8-install`
  - If applicable, replace the behaviors of the low level display device interface `defer` words with more appropriate behavior. (See the next section for details.)

- If necessary, correct the centering of the image on the screen by changing the value of `window-left` and / or `window-top`.

When `is-install` (-- xt) is executed, it creates the following methods:

- **open** (-- ok?)

  When later called, executes the routine whose execution token is *xt* guarded by `catch`. If the execution of *xt* results in a `throw`, `false` is returned. Otherwise, the Open Firmware terminal emulator is initialized and `true` is returned.

- **write** (addr len -- #written)

  When later called, passes its argument string to the Open Firmware terminal emulator for interpretation.

- **draw-logo** (line# addr width height --)

  When later called, executes the routine whose execution token was installed in the `defer` word `draw-logo`. Initially, `fb8-install` loads `draw-logo` with the behavior of `fb8-draw-logo`. (See the next section for more details.)

- **restore** (--)

  When later called, executes the routine whose execution token was installed in the `defer` word `reset-screen`. Initially, `fb8-install` loads `reset-screen` with the behavior of `fb8-reset-screen`. (See the next section for more details.)

Low Level Display Device Interface `defer` Words

The low level display device interface is composed of the following `defer` words:

- `draw-character`
- `reset-screen`
- `toggle-cursor`
- `erase-screen`
- `blink-screen`
- `invert-screen`
- `insert-characters`
- `delete-characters`
- `insert-lines`
- `delete-lines`
- `draw-logo`

When `fb8-install` is executed, each of these words is loaded with a default behavior supplied by the `fb8` default versions of these words (i.e. `draw-character` is loaded with the behavior of `fb8-draw-character`).

If your hardware is capable of performing a given operation more efficiently than one of the default methods, you may create an alternative method in FCode to take advantage of your hardware's capabilities, and may then replace the default behavior with your alternative method.

For example, your hardware might have the capability of quickly erasing the screen. If you wrote a word named (say) `my-erase-screen`, you could replace the default behavior with the phrase:

```
['] my-erase-screen  to erase-screen
```

The complete definitions of the `defer` words can be found in Chapter 12 “Open Firmware Dictionary”.

`is-remove` Actions

The word whose execution token is installed with `is-remove` must:

- Reset the hardware, if applicable.
- Unmap any mapped resources.
- Disable PCI memory and/or PCI I/O space accesses as appropriate.
- Delete the `address` property.

When `is-remove` (-- xt) is executed, it creates the following method:

- **close** (--)

  When later called, executes the routine whose execution token is *xt*.

`is-selftest` Actions

The word whose execution token is installed with `is-selftest` must:

- Must assume that the device may or may not be open.
- Establish any device state necessary to perform its tests.
- Perform a selftest of the display device returning 0 on test success and a non-zero error code on test failure. The complexity of this test may depend upon the value returned by `diagnostic-mode?`; if so, more extensive testing should be done when `diagnostic-mode?` returns `true`.
- Release any resources that were allocated to perform the tests.

When `is-selftest` (-- xt) is executed, it creates the following method:

- **selftest** (-- failure?)

  When later called, executes the routine whose execution token is *xt*.

`display` Device Driver Issues

16-Color Text Extension Recommended Practice

IEEE Standard 1275-1994 defines the facilities for displaying text in terms of a two color model. Most computers today have color capability. The *16-Color Text Extension Recommended Practice* describes extensions to the basic Terminal Emulator support package that enable the use of additional colors on the Open Firmware console `display` device.

A `display` driver that uses the `fb8` support routines only has to meet two additional requirements to support this extension:

- Define the property `iso6429-1983-colors`.

- Set up the device’s color translation mechanism such that the correspondence between pixel values in the frame buffer memory and displayed colors is as given in Table 30.

Table 30 16 Color Text Extension Color Assignments

Color Number	Red	Green	Blue	Color
0	0	0	0	Black
1	0	0	2/3	Blue
2	0	2/3	0	Green
3	0	2/3	2/3	Cyan
4	2/3	0	0	Red
5	2/3	0	2/3	Magenta
6	2/3	1/3	0	Brown
7	2/3	2/3	2/3	White
8	1/3	1/3	1/3	Grey
9	1/3	1/3	1	Bright Blue
10	1/3	1	1/3	Bright Green
11	1/3	1	1	Bright Cyan
12	1	1/3	1/3	Bright Red
13	1	1/3	1	Bright Magenta
14	1	1	1/3	Yellow
15	1	1	1	Bright White

The mechanism for creating this correspondence is the device driver’s responsibility. For those devices having a color lookup table, loading the first sixteen entries as shown above should achieve this result.

Note – The above table defines the sixteen colors in terms of the approximate intensities of red, green and blue, where 0 means “no intensity” and 1 means “maximum intensity”. The description of the colors does not imply that the hardware must use an RGB color space.

Note – Some Sun system ROM implementations use color number 255 (i.e. 0xFF) as the background color. To make your driver compatible with those systems, also load color number 255 with the values shown above for color number 15.

If your device cannot support 8-bit frame buffers (e.g. has a 24-bit-only frame buffer), the device driver must provide additional capabilities that would otherwise have been provided by the `fb8` support package. See the *16-Color Text Extension Recommended Practice* document for the details.

8-Bit Graphics Extension

IEEE Standard 1275-1994 defines a text oriented interface. Most computers today have graphics capabilities, and users generally prefer graphical user interfaces over command line interfaces like that defined by *IEEE Standard 1275-1994*. The *8-Bit Graphics Extension Recommended Practice* document describes extensions to the standard that enable the manipulation of graphical, 256-color objects on `display` devices.

The graphics model used by this extension has the following characteristics:

- Pixels are represented by 8-bit values that represent one of 256 colors.
- The mapping of a color number to a display color is specified with three 8-bit values each of which represent a color component in an RGB color space.
- A color component value of 0 represents the absence of that color while a value of 255 represents full saturation of the color (i.e. [0, 0, 0] is black and [255, 255, 255] is white).
- When a color is specified with a memory region (as with `set-colors` and `get-colors`), the first byte of the region represents the red component, the next byte represents the green component and the next byte represents the blue component. If multiple colors are specified with a memory region, the byte describing the red component of the *(N+1)*th color immediately follows the byte describing the blue component of the *N*th color.
- The top-left corner of the display is [0, 0].
- Rectangular regions of the display buffer are described by a set of coordinates specifying the position of the top-left corner of the rectangle [*x*, *y*] and the width and height of the rectangle [*w*, *h*].
- Data in memory representing rectangular regions consist of *w* times *h* contiguous bytes where the first *w* bytes represent the pixels of the first row of the rectangle (from left to right), the next *w* bytes represent the pixels of the second row, etc. Each such byte contains the color number of the corresponding pixel.

This extension defines the following methods that must be added to the `display` device driver.

draw-rectangle (addr x y w h --)
Display the rectangular image beginning at pixel location *x,y* of size *w,h* using the image defined by the memory region starting at *addr*.

fill-rectangle (index x y w h --)
Fill the rectangular region beginning at pixel location *x,y* of size *w,h* using the color specified by *index*.

read-rectangle (addr x y w h --)
Copy the rectangular image beginning at pixel location *x,y* of size *w,h* to the memory region starting at *addr*.

Note – For displays that are not in 8-bit per pixel mode, `read-rectangle` is not defined. It is therefore recommended that displays provide an 8-bit mode and use this mode during Open Firmware execution.

---

color! (r g b index --)
Set the color associated with *index* to the value specified by *r,g,b*.

color@ (index -- r g b)
Read the color associated with *index* and return its *r,g,b* components.

set-colors (addr index #colors --)
Set a range of *#colors* consecutive colors starting at *index*. *addr* is the address of the memory area in which the color components are specified.

get-colors (addr index #colors -- **)**
Read a range of *#colors* consecutive colors starting at *index*. *addr* is the address of the memory area into which the color components are copied.

dimensions (-- width height)
Return the dimensions in pixels of the viewable area of the screen in the current mode.

Use of Legacy VGA Addressing

There are a couple of issues associated with the use of VGA ISA legacy addresses in I/O space.

VGA I/O space registers are non-relocatable and are distributed in small regions across I/O space. Strictly speaking, each of these small regions should be independently mapped. However, given that these addresses in non-relocatable I/O space are not going to be changed, it is safe—and therefore acceptable—to map in one large region that covers all of the registers used by the driver.

Having said that, it is important to accurately report in the `reg` property the actual registers decoded by the card with as many entries as are required. The following code example uses four separate `reg` property entries to report the registers decoded by a generic VGA card.

Since all VGA cards use these same non-relocatable legacy addresses, if two devices have these addresses mapped simultaneously then both devices will respond to accesses to the addresses. Consequently, it is good practice to “wrap” accesses to the legacy addresses in a `map-in`/`map-out` pair such that the legacy addresses are only “consumed” by the driver for a short period of time. Failure to do this may result in problems if two copies of your device are ever installed in the same system.

Device Driver Example

Generic VGA Display Device Driver

This example FCode program is a complete bootable generic VGA console display device driver.

Code Example 10-1 Complete Generic VGA Display Device Driver

```
\ Complete Generic VGA Display Device Driver
fcode-version2
hex

: copyright  ( -- )
   ." Copyright (c) 1994-1996 FirmWorks.  All Rights Reserved." cr
;
```

Code Example 10-1 Complete Generic VGA Display Device Driver *(Continued)*

```
headers

-1 instance value io-base

: config-w@  ( config-addr -- data )  " config-w@" $call-parent  ;
: config-w!  ( data config-addr -- )  " config-w!" $call-parent  ;

: map-io-regs  ( -- )
   \ h# 8100.0000 means non-relocatable I/O space
   0 0 h# 8100.0000  h# 1.0000  " map-in" $call-parent to io-base

   \ Enable I/O space response
   my-space 4 +  dup   config-w@  ( addr value )
   1 or          swap  config-w!
;
: unmap-io-regs  ( -- )
   \ Disable I/O space response
   my-space 4 +  dup   config-w@  ( addr value )
   1 invert and  swap  config-w!

   io-base  h# 1.0000  " map-out"  $call-parent
   -1 to io-base
;

: pc@  ( offset -- byte )  io-base +  rb@  ;
: pc!  ( byte offset -- )  io-base +  rb!  ;
: pw!  ( word offset -- )  io-base +  rw!  ;

\ Access functions for various register banks

\ Reset attribute address flip-flop
: reset-attr-addr  ( -- )  3da ( input-status1 )  pc@ drop  ;

: setup-vse!  ( b -- )  46e8 pc!  ;

: dac@  ( -- b )  3c8 pc@  ;

: video-mode!  ( b -- )  reset-attr-addr  03c0 pc!  ;
: attr!  ( b index -- )  03c0 pc!  03c0 pc!  ;
: attr@  ( index -- b )
   reset-attr-addr  03c0 pc!  03c1 pc@  reset-attr-addr
;
: grf!   ( b index -- )  03ce pc!  03cf pc!  ;

: feature-ctl!  ( b -- )  03da pc!  ;

: misc@  ( -- b )  3cc pc@  ;
: misc!  ( b -- )  3c2 pc!  ;

: crt-setup  ( index -- data-addr )  03d4 pc!  03d5  ;
: crt!  ( b index -- )  crt-setup pc!  ;
: crt@  ( index -- b )  crt-setup pc@  ;
: crt-data!  ( b -- )  03d5 pc!  ;
: crt-set   ( bits index -- )  crt@  or  crt-data!  ;
: crt-clear ( bits index -- )  crt@  swap invert and  crt-data!  ;
```

Code Example 10-1 Complete Generic VGA Display Device Driver *(Continued)*

```
: seq-setup  ( index -- data-addr )  03c4 pc!  03c5  ;
: seq!  ( b index -- )  seq-setup pc!  ;
: seq@  ( index -- b )  seq-setup pc@  ;

: unlock  ( -- )        \ Unlock all registers
   80 11 crt-clear      \ Unlock CRT registers
   80  3 crt-set        \ Unlock vertical retrace registers
;
: wakeup  ( -- )
   1e  setup-vse!   \ Video system enable, in setup mode
   1  102  pc!      \ Enable VGA video subsystem
   e  setup-vse!    \ Out of setup mode
   23 3c2  pc!      \ Enable memory, color base, page 0, clock @ 25.175 MHz
   unlock
;

: async-reset  ( -- )
   20 1 seq!                    \ screen off
    0 0 seq!    3 0  seq!       \ Pulse reset
   1 seq@ 20 invert and  1 seq! \ Screen on
;

: low-power  ( -- )
   ff  4 crt!           \ Disable hsync for low monitor power
;

\ Standard VGA CRT Controller registers, indices 0-h#18
: crt-table  ( -- addr len )  \ 72 Hz
   " "(5f 4f 50 82 54 80 bf 1f 00 41 00 00 00 00 00 31 9c 0e 8f 28 40 96 b9 a3 ff)"
;
: crt-regs  ( -- )
   crt-table  0  ?do
      i 4 <>  if  \ Don't program hsync (at offset 4) until later
         dup i + c@  i  crt!
      then
   loop
   drop
;

: attr-table  ( -- addr len )    \ Attribute controller indices 0-14
   " "(00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 41 00 0f 00 00)"
;
: attr-regs  ( -- )
   reset-attr-addr
   attr-table 0  do  dup i + c@  i attr!  loop  drop
;

: grf-table  ( -- addr len )     \ Graphics controller indices 0-8
   " "(00 00 00 00 00 40 05 0f ff)"
;
: grf-regs  ( -- )
   grf-table  0  do  dup i + c@  i grf!   loop  drop
;
```

```
: seq-table  ( -- addr len )  " "(01 0f 00 0e)"  ;
: seq-regs  ( -- )
   seq-table  0  ?do  dup i + c@  i 1+ seq!  loop  drop
;

external
\ Set color lookup table to comply with 16-Color Text Extension Recommended Practice
: default-colors  ( -- addr index #indices )
   \ The following array must be entered as one long line of text.
   " "(00 00 00  aa 00 00  00 aa 00  aa 55 00  00 00 aa  aa 00 aa  00 aa aa  aa aa aa
55 55 55  ff 55 55  55 ff 55  ff ff 55  55 55 ff  ff 55 ff  55 ff ff  ff ff ff)"
   0 swap 3 /
;

headers
\ Palette access
: init-dac  ( -- )
   ff  03c6 pc!
;

: index!  ( index -- )  03c8 pc!  ;
: plt!  ( b -- )  03c9 pc!  ;
: plt@  ( -- b )  03c9 pc@  ;

external
\ Methods defined by the 8-Bit Graphics Extension Recommended Practice
: set-colors  ( addr index #indices -- )
   swap index!
   3 *  bounds  ?do  i c@  plt!  loop
;
: get-colors  ( addr index #indices -- )
   swap index!
   3 *  bounds  ?do  plt@  i c!  loop
;
: color@  ( index -- r g b )  index!  plt@ plt@ plt@  ;
: color!  ( index -- r g b )  index!  swap rot  plt!  plt!  plt!  ;

headers

: setup-middle  ( -- )
   low-power

   0 video-mode!            \ Disable video palette
   async-reset

   6 4 seq!                 \ Enable access to all 256K of VGA memory

   seq-regs

   attr-regs
   grf-regs

   crt-regs

   0 feature-ctl!           \ Vertical sync ctl
;
```

Code Example 10-1 Complete Generic VGA Display Device Driver *(Continued)*

```
: hsync-on  ( -- )
   crt-table drop 4 + c@ 4 crt!   \ Set hsync position
;

: setup-end  ( -- )
   init-dac
   hsync-on
   default-colors set-colors
   ff dup dup dup color!   \ Load Bright White in color 255 for compatibility with Sun

   20 video-mode!          \ Video on
;

: setup-begin  ( -- )  wakeup  ;

: init  ( -- )
   \ Apparently the clocks take awhile to stabilize, so it is
   \ sometimes necessary to do the setup twice
   setup-begin
   setup-middle
   setup-end
;

" display" device-name
" FirmWorks,generic-vga"  model
" display"  device-type
" ISO8859-1" encode-string  " character-set"  property
0 0  encode-bytes  " iso6429-1983-colors"  property

d# 320 constant /scanline
d# 200 constant #scanlines
/scanline #scanlines * constant /fb

\Define reg property
\ PCI Configuration Space Registers
my-address my-space  encode-phys   0 encode-int encode+  0 encode-int encode+

\ Memory Space Base Address Register 10
\ Despite what the configuration base address register implies,
\ the S3’s memory region is mappable on 2 Mbytes boundaries.
\ This is a violation of the PCI spec, which requires that the base
\ address register must accurately describe the mapping granularity.

my-address my-space h# 200.0010 or encode-phys encode+
0 encode-int encode+ h# 20.0000 encode-int encode+

\ PCI Expansion ROM
my-address my-space h# 200.0030 or encode-phys encode+
0 encode-int encode+ h# 10.0000 encode-int encode+

\ VGA Sleep Register
h# 102 0  my-space h# a100.0000 or encode-phys encode+
0 encode-int encode+ h# 1 encode-int encode+

\ VGA Monochrome Emulation Mode Registers
h# 3b0 0  my-space h# a100.0000 or encode-phys encode+
0 encode-int encode+ h# c encode-int encode+
```

Code Example 10-1 Complete Generic VGA Display Device Driver *(Continued)*

```
\ VGA Control and Color Emulation Mode Registers
h# 3c0 0  my-space h# a100.0000 or encode-phys encode+
0 encode-int encode+ h# 20 encode-int encode+

\ VGA Subsystem Enable Register
h# 46e8 0  my-space h# a100.0000 or encode-phys encode+
0 encode-int encode+ h# 1 encode-int encode+

\ Frame Buffer
h# a.0000 0  my-space h# a100.0000 or encode-phys encode+
0 encode-int encode+ h# 2.0000 encode-int encode+
" reg" property

\ Some of Apple's Open Firmware implementations have a bug in their map-in method. The
\ bug causes phys.lo and phys.mid to be treated as absolute addresses rather than
\ offsets even when working with relocatable addresses.

\ To overcome this bug, the Open Firmware Working Group in conjunction with Apple has
\ adopted a workaround that is keyed to the presence or absence of the add-range method
\ in the PCI node. If the add-range method is present in an Apple ROM, the map-in
\ method is broken. If the add-range property is absent, the map-in method behaves
\ correctly.

\ The following methods allow the FCode driver to accomodate both broken and working
\ map-in methods.

: map-in-broken?  ( -- flag )
   \ Look for the method that is present when the bug is present
   " add-range"  my-parent  ihandle>phandle   ( adr len phandle )
   find-method  dup  if  nip  then            ( flag )  \ Discard xt if present
;

\ Return phys.lo and phys.mid of the address assigned to the PCI base address
\ register indicated by phys.hi .
: get-base-address  ( phys.hi -- phys.lo phys.mid phys.hi )
   " assigned-addresses" get-my-property  if   ( phys.hi )
      ." No address property found!" cr
      0 0 rot  exit                            \ Error exit
   then                    ( phys.hi adr len )

   rot >r                  ( adr len )  ( r: phys.hi )
   \ Found assigned-addresses, get address
   begin  dup  while       ( adr len' )  \ Loop over entries
      decode-phys          ( adr len' phys.lo phys.mid phys.hi )
      h# ff and  r@ h# ff and  =  if  ( adr len' phys.lo phys.mid )  \ This one?
         2swap 2drop       ( phys.lo phys.mid )          \ This is the one
         r> exit           ( phys.lo phys.mid phys.hi )
      else                 ( adr len' phys.lo phys.mid ) \ Not this one
         2drop             ( adr len' )
      then                 ( adr len' )
      decode-int drop decode-int drop        \ Discard boring fields
   repeat
   2drop                   ( )
   ." Base address not assigned!" cr

   0 0 r>                  ( 0 0 phys.hi )
;
```

Code Example 10-1 Complete Generic VGA Display Device Driver *(Continued)*

```
: map-frame-buffer    ( -- )
   \ Map frame buffer
   map-in-broken?  if
      my-space h# 8200.0010 or  get-base-address   ( phys.lo phys.mid phys.hi )
   else
      my-address my-space h# 200.0010 or           ( phys.lo phys.mid phys.hi )
   then                                            ( phys.lo phys.mid phys.hi )

   /fb " map-in" $call-parent
   a.0000 +                     \ Generic VGA compatible address
   to frame-buffer-adr

   \ Enable memory space access
   my-space 4 +  dup  config-w@  ( addr value )
   2 or  swap  config-w!

   frame-buffer-adr encode-int " address" property
;

: unmap-frame-buffer  ( -- )
   frame-buffer-adr a.0000 - /fb " map-out" $call-parent
   -1 to frame-buffer-adr

   \ Disable memory space access
   my-space 4 +  dup  config-w@  ( addr value )
   2 invert and  swap  config-w!

   " address" delete-property
;

: display-install  ( -- )
   map-io-regs  init
   map-frame-buffer   default-font set-font
   /scanline #scanlines over char-width / over char-height / fb8-install
;

: display-remove  ( -- )  unmap-frame-buffer  unmap-io-regs  ;

/scanline  encode-int " width"     property
#scanlines encode-int " height"    property
8          encode-int " depth"     property
/scanline  encode-int " linebytes" property

['] display-install  is-install
['] display-remove   is-remove

external

\ Methods defined by the 8-Bit Graphics Extension Recommended Practice
: fill-rectangle  ( index x y w h -- )
   2swap  -rot  /scanline *  +  frame-buffer-adr +    ( index w h fbadr )
   swap  0  ?do                                       ( index w fbadr )
      3dup swap rot fill                              ( index w fbadr )
      /scanline +                                     ( index w fbadr' )
   loop
   3drop
;
```

Code Example 10-1 Complete Generic VGA Display Device Driver *(Continued)*

```
: draw-rectangle  ( addr x y w h -- )
   2swap  -rot  /scanline *  +  frame-buffer-adr +    ( addr w h fbadr )
   swap  0  ?do                                       ( addr w fbadr )
      3dup swap move                                  ( addr w fbadr )
      >r  tuck + swap  r>                             ( addr' w fbadr )
      /scanline +                                     ( addr' w fbadr' )
   loop
   3drop
;

: read-rectangle  ( addr x y w h -- )
   2swap  -rot  /scanline *  +  frame-buffer-adr +    ( addr w h fbadr )
   swap  0  ?do                                       ( addr w fbadr )
      3dup -rot move                                  ( addr w fbadr )
      >r  tuck + swap  r>                             ( addr' w fbadr )
      /scanline +                                     ( addr' w fbadr' )
   loop
   3drop
;

: dimensions  ( -- width height )  /scanline #scanlines  ;

fcode-end
```

11 Memory-Mapped Buses

A *memory-mapped bus* logically extends the processor's memory address space to include the devices on that bus. This enables the physical addresses of the children of the bus device to be mapped into the CPU's virtual address space and to be directly accessed like memory using processor load and store cycles.

Memory-mapped buses include such buses as PCI, SBus and VMEbus.

Not all bus devices fall into this category. For example, SCSI is not a memory-mapped bus; SCSI targets are not accessed with load or store instructions.

Required Methods

A memory-mapped bus package code must implement the `open`, `close`, `reset`, and `selftest` methods, as well as the following:

decode-unit (addr len -- phys.lo … phys.hi)

Convert *addr len*, a text string representation, to *phys.lo … phys.hi*, a numerical representation of a physical address within the address space defined by this package. The format of *phys.lo … phys.hi* varies from bus to bus.

dma-alloc (… size -- virt)

Allocate a virtual address range of length *size* bytes that is suitable for direct memory access by a bus master device. The memory is allocated according to the most stringent alignment requirements for the bus. *virt* is an 32-bit address that the Open Firmware-based system can use to access the memory.

Note that `dma-map-in` must also be called to generate a suitable DMA address.

A child of a memory-mapped device calls `dma-alloc` using

```
  " dma-alloc" $call-parent
```

For example:

```
-1 value my-vaddr \ Conventionally set to -1 indicating an invalid
                  \ virtual address
: my-dma-alloc ( size -- )
   " dma-alloc"  $call-parent  to my-vaddr
;
```

dma-free (virt size **--**)

Free *size* bytes of memory previously allocated by `dma-alloc` at the virtual address *virt*.

A child of a memory-mapped device calls `dma-free` by using

```
 " dma-free" $call-parent
```

For example:

```
2000 value my-size
: my-dma-free  ( -- )
   my-vaddr my-size " dma-free"  $call-parent
   -1 to my-vaddr
;
```

dma-map-in (… virt size cacheable? **--** devaddr)

Convert the virtual address range *virt size*, previously allocated by `dma-alloc`, into an address *devaddr* suitable for DMA on the bus. `dma-map-in` can also be used to map application-supplied data buffers for DMA use if the bus allows. If *cacheable?* is true, the calling child desires to use any available fast caches for the DMA buffer. If access to the buffer is required before the buffer is mapped out, the child must call `dma-sync` or `dma-map-out` to ensure cache coherency with memory.

A child of a memory-mapped device calls `dma-map-in` using

```
 " dma-map-in" $call-parent
```

For example:

```
: my-vaddr-dma-map ( -- )
   my-vaddr my-size false " dma-map-in"  $call-parent   ( devaddr )
   to my-vaddr-dma
;
```

dma-map-out (virt devaddr size **--**)

Remove the DMA mapping previously created with `dma-map-in`. Flush all caches associated with the mapping.

A child of a memory-mapped device calls dma-map-in by using

```
 " dma-map-out" $call-parent
```

For example:

```
$call-parent
: my-vaddr-dma-free ( -- )
   my-vaddr my-vaddr-dma my-size " dma-map-out"  $call-parent
   -1 to my-vaddr-dma
;
```

dma-sync (virt devaddr size --)

Synchronize (flush) any memory caches associated with the DMA mapping previously established by `dma-map-in`. You must interleave calls to this method (or `dma-map-out`) between DMA and CPU accesses to the memory region, or you may not obtain the most recent data written into the cache.

For example:

```
: my-dma-sync ( virt devadr size -- )
   " dma-sync" $call-parent
;
```

probe-self (arg-str arg-len reg-str reg-len fcode-str fcode-len --)

Probe for a child of this node. *fcode-str fcode-len* is a unit-address text string that identifies the location of the FCode program for the child. *reg-str reg-len* is a probe-address text string that identifies the address of the child itself. *arg-str arg-len* is an instance-arguments text string for any device arguments for the child (which can be retrieved within the child's FCode program with the `my-args` FCode). `probe-self` checks whether there is indeed FCode at the indicated location, perhaps by mapping the device and using `cpeek` to ensure that the device is present and that the first byte is a valid FCode start byte.

If the FCode exists, `probe-self` creates a new child device node and interprets the FCode. If the interpretation of the FCode fails in some way, the new device node may be empty, containing no properties or methods.

For example, to probe FCode for PCI Device 0:

```
" /pci" open-dev
0 0 " 0" 2dup probe-self
device-end
```

map-in (phys.lo … phys.hi size -- virt)

Create a mapping associating the range of physical addresses beginning at *phys.lo … phys.hi* extending for *size* bytes within the package's physical address space with a processor virtual address *virt*.

The number of cells in the list *phys.lo … phys.hi* is determined by the value of the `"#address-cells"` property of the node containing `map-in`.

For example, a child of a memory-mapped device calls `map-in` with `" map-in" $call-parent`. (The following example assumes that the value of the parent's `"#address-cells"` property is 3):

```
: map-reg ( -- )
   my-address xx-offset 0 d+ my-space  ( phys.lo phys.mid phys.hi )
   xx-size " map-in" $call-parent      ( virt )
   to xx-vaddr                         ( )
;
```

map-out (virt size --)

Destroy the mapping set by `map-in` at virtual address virt of length *size* bytes.

For example, a child of a memory-mapped device calls `map-out` with `" map-out" $call-parent`:

```
: unmap-reg ( -- )
   xx-vaddr xx-size           ( virt size )
   " map-out" $call-parent    ( )
   -1 to xx-vaddr
;
```

PCI Bus Addressing

The PCI Bus has three distinct address spaces: Configuration, Memory and I/O.

Configuration space addressing is geographical addressing with numbered buses, devices, functions and registers. Memory space allows for up to 64 bit addressing. I/O space allows for up to 32 bit addressing.

A PCI address is represented numerically with three, 32-bit cells, *phys.hi*, *phys.mid* and *phys.lo*. The text representation may take any of 5 different forms. Please refer to *PCI Bus Binding to IEEE Standard 1275-1994* for a detailed description.

PCI Required Properties

Table 31 Required PCI Properties

Property Name	Sample Value
name	" AAPL,finagle"
device_type	" pci"
#address-cells	3
#size-cells	2
reg	
ranges	
clock-frequency	
bus-range	
slot-names	
bus-master-capable	

SBus Addressing

The SBus uses geographical addressing with numbered slots.

An SBus physical address is represented numerically as two numbers, *phys.hi* and *phys.lo*. *phys.hi* contains the SBus slot number and *phys.lo* contains the offset from the base of that slot.

The text string representation is *slot#, offset* where both *slot#* and *offset* are the ASCII representations of hexadecimal numbers. *slot#* encodes *phys.hi* and *offset* encodes *phys.lo*.

Please refer to *IEEE Draft Std P1275.2/D14a Standard for Boot (Initialization Configuration) Firmware Supplement for IEEE 1496 (SBus) Bus* for a detailed description.

SBus Required Properties

Table 32 Required SBus Properties

Property Name	Sample Value
`name`	`" SUNW,finagle"`
`device_type`	`" sbus"`
`ranges`	
`reg`	
`burst-sizes`	
`clock-frequency`	
`slot-address-bits`	

VMEBus Addressing

VMEBus has a number of distinct address spaces represented by a subset of the 64 possible values encoded by the six “address modifier” bits. The maximum size of one of these address spaces is 32 bits. An additional bit is used to select between 16-bit and 32-bit data.

A VMEBus physical address is represented numerically as two numbers, *phys.hi* and *phys.lo*. *phys.hi* is made up of the six address modifier bits AM0-5 in bits 0-5 and the data width bit (0 = 16-bit data, 1 = 32-bit data) in bit 6. *phys.lo* is the offset within the selected address space.

The text string representation is *AML,VME-address* where both *AML* and *VME-address* are ASCII representations of hexadecimal numbers. *AML* encodes *phys.hi* and *VME-address* encodes *phys.lo*.

Please refer to *IEEE Draft Std P1275.3/D8 Standard for Boot (Initialization Configuration) Firmware Supplement for IEEE 1014-1987 (VME) Bus* for a detailed description.

VMEBus Required Properties

Table 33 Required VMEbus Properties

Property Name	Sample Value
`name`	`" SUNW,vizzy"`
`device_type`	`" vmebus"`
`ranges`	
`reg`	

12
Open Firmware Dictionary

This dictionary describes all of the words defined by *IEEE Standard 1275-1994*. Included within this dictionary are all of the pre-defined FCode words that you can use as part of FCode source code programs. Appendix A, “FCode Reference”, contains a command summary, with words grouped by function.

The dictionary also includes assembler directives, debugger commands, tokenizer directives and macros, configuration variables, properties, standard methods, `nvedit` commands, Client Interface commands and User Interface commands.

The words are given alphabetically in this chapter, sorted by the first alphabetic character in the word’s name. For example, the words `mod` and `*/mod` are adjacent to each other. Words having no alphabetic characters in their names are placed at the beginning of the chapter, in ASCII order.

The boot ROM and tokenizer are case-insensitive (all Forth words are converted to lowercase internally). The only exceptions are literal text, such as text inside " strings and text arguments to the `ascii` command, which are left in the original form. In general, you may use either uppercase or lowercase. By convention, Open Firmware drivers are written in lowercase.

All arithmetic uses 32-bit signed values, unless otherwise specified.

Defining words create a header by calling `external-token`, `named-token`, or `new-token`. See the definitions of these words for more details.

All FCode byte values listed in this chapter are given in hexadecimal.

The stack diagram notation used in this chapter is described by Table 2, “Stack Item Notation,” on page 9.

The dictionary definitions have the following form:

name	“pronunciation”
stack:	(stack diagram)
code:	FCode#
generates:	tokenizer macro (if applicable)

Prose description.

! “store”

stack: (x a-addr --)
code: 72

Stores x at *a-addr*. For more portable code, use `l!` if you explicitly want a 32-bit access. *a-addr* must be aligned as given by `variable`.

See also: `c!`, `w!`, `l!`, `rb!`, `rw!`, `rl!`

" “quote”

stack: ([text<">< >] -- text-str text-len)
code: none
generates: `b(")` `len-byte` `xx-byte` … `xx-byte`

Gathers the immediately following text string or hex data until reaching the terminator `"<whitespace>`.

At execution time, the address and length of the string is left on the stack. For example:

```
  " AAPL,new-model" encode-string " model" property
```

You can embed control characters and 8-bit binary numbers within strings. This is similar in principle to the `\n` convention in C, but syntactically tuned for Forth. This feature applies to the string arguments of the words `"` and `."`

The escape character is ‘`"`’. The list of escape sequences is:

Table 34 Escape Sequences in Text Strings

Syntax	Function
`""`	quote (")
`"n`	newline
`"r`	carriage return
`"t`	tab
`"f`	formfeed
`"l`	linefeed
`"b`	backspace
`"!`	bell
`"^`*x*	control *x*, where *x* is any printable character
`"(hh hh)`	Sequence of bytes, one byte for each pair of hex digits `hh` . Non-hex characters will be ignored

`"` followed by any other printable character not mentioned above is equivalent to that character.

`"(` means to start parsing pairs of hexadecimal digits as one or more 8-bit characters in the range 0x00 through 0xFF, delimited by a trailing `)` and ignoring non-hexadecimal digits between pairs of hexadecimal digits. Both uppercase and lowercase hexadecimal digits are recognized. Since non-hex characters (such as space or comma) are ignored between `"(` and `)`, these characters make useful delimiters. (The “makearray” tool can be used in conjunction with this syntax to easily incorporate large binary data fields into any FCode Program.)

Any characters thus recognized are appended to any previous text in the string being assembled. After the `)` is recognized, text assembly continues until a trailing `"<whitespace>`.

For example:

```
" This is "(01 32,8e)abc"nA test xyzzy "!"! abcdefg""hijk"^bl"
      ^^^^^^^^   ^        ^ ^      ^    ^
      3 bytes    newline    2 bells   "    control b
```

Note – The use of `"n` for line breaks is discouraged. The preferred method is to use `cr`, rather than embedding the line break character inside a string. Use of `cr` results in more accurate display formatting, because Forth updates its internal line counter when `cr` is executed.

When " is used outside a colon definition, only two interpreted strings of up to 80 characters each can be assembled concurrently. This limitation does not apply in colon definitions.

See also: `b(")`

#

stack: (ud1 -- ud2)
code: C7

Converts a digit *ud1* in pictured numeric output conversion. Typically used between <# and #>.

#>

stack: (ud -- str len)
code: C9

Ends pictured numeric output conversion. *str* is the address of the resulting output array. *len* is the number of characters in the output array. *str* and *len* together are suitable for `type`. See `(.)` and `(u.)` for typical usages.

' “tick”

stack: ("old-name< >" -- xt)
code: none
generates: `b(') old-FCode#`

Generates the execution token (*xt*) of the word immediately following `'` in the input stream. `'` should only be used *outside* of definitions. See `b('), [']` for more details.

For example:

```
defer opt-word   ( -- ) ' noop is opt-word
```

(

stack: ([text<)> --)
code: none

Causes the compiler/interpreter to ignore subsequent text after the "`(` " up to a delimiting "`)`" . Note that a space is required after the `(`. Although either `(` or `\` may be used equally well for documentation, by common convention we use (…) for stack comments and \ … for all other text comments and documentation.

For example:

```
: 4drop             ( a b c d -- )
   2drop            ( a b )
   2drop            ( )
;
```

See also: (s

(.)

stack: (n -- str len)
code: none
generates: dup abs <# u#s swap sign u#>

Converts a number into a text string according to the value in base.This is the numeric conversion primitive, used to implement display words such as "." If *n* is negative, the first character in the array will be a minus (-) sign.

For example:

```
" CPU boot: show-version ( -- )
   .rom version is " base @  d# 16 base !  ( old-base )
   firmware-version  ( old-base version )
   lwsplit (.) type ascii . emit .h cr base !         ( )
```

***** “star”

stack: (nu1 nu2 -- prod)
code: 20

prod is the arithmetic product of *nu1* times *nu2*. If the result cannot be represented in one stack entry, the least significant bits are kept.

***/** “star slash”

stack: (n1 n2 n3 -- quot)
code: none

Calculates *n1*n2/n3*. The inputs, outputs and intermediate products are all 32-bit.

+ “plus”

stack: (nu1 nu2 -- sum)
code: 1E

sum is the arithmetic sum of *nu1* plus *nu2*.

+! “plus store”

stack: (nu a-addr --)
code: 6C

nu is added to the value stored at a-*addr*. This sum replaces the original value at *a-addr*. *a-addr* must be aligned as given by variable.

, “comma”

stack: (x --)

code: D3

Reserves one cell of storage in data-space and stores *x* in the cell.The data space pointer must be aligned prior to the execution of `,`.

For example, to create an array containing integers 40004000 23 45 6734:

```
create my-array 40004000 , 23 , 45 , 6734 ,
```

- “minus”

stack: (nu1 nu2 -- diff)

code: 1F

diff is the result of subtracting *nu1* minus *nu2*.

. “print”

stack: (nu --)

code: 9D

Displays the absolute value of *nu* in a free field format with a leading minus sign if *nu* is negative, and a trailing space.

If the base is hexadecimal, `.` displays the number in unsigned format, since signed hex display is hardly ever wanted. Use `s.` to display signed hex numbers.

See also: `s.`, `.d`, `.h`

." “dot quote”

stack: ([text<">] --)

code: none

generates: `b(") len text type`

This word compiles a text string, delimited by `"<whitespace>` e.g. `." hello world"` .

At execution time, the string is displayed. This word is equivalent to using `" text" type` .

`."` is normally used only within a definition. The text string will be displayed later when that definition is called. You may wish to follow it with `cr` to flush out the text buffer immediately. Use `.(` for any printing to be done immediately.

See also: `"`, `.(`, `tokenizer[`

.(

stack: ([text<)>] --)

code: none

Gathers a text string, delimited by `)` , to be immediately displayed. For example:

```
.( hello world)
```

This word is equivalent to: `" text"  type`

Use `.(` to print out text immediately. (You should follow it with a `cr` to flush out the text buffer immediately). `.(` may be called either inside or outside of definitions; the text is immediately displayed in either case.

Note that during FCode interpretation the string will typically be printed out of serial port A, since any framebuffer present may not yet be activated at the time that PCI slots are being probed. Use `."` for any printing to be done when new words are later executed.

See also: `."`, `tokenizer[`

/

stack:	(n1 n2 -- quot)
code:	21

Calculates *n1* divided by *n2*. An error condition results if the divisor (*n2*) is zero. See `/mod`.

"/"

The root node of the device tree.

:

stack:	(E: … -- ???)
	(C: "new-name< >" -- colon-sys)
code:	none
generates:	`new-token|named-token|external-token b(:)`

Begins a new definition, terminated by `;` Used in the form:

```
: my-newname … ;
```

Later usage of `my-newname` is equivalent to usage of the contents of the definition.

See `named-token`, `new-token`, and `external-token` for more information on header formats.

;

stack:	(C: colon-sys --)
	(--) (R: -- nest-sys)
code:	none
generates:	`b(;)`

Ends the compilation of a colon definition. Upon later execution, returns control to the calling definition specified by *nest-sys*.

See also: `:`

<

stack:	(n1 n2 -- less_than?)
code:	3A

less_than? is true if *n1* is less than *n2*. *n1* and *n2* are signed integers.

<#

stack: (--)
code: 96

Initializes pictured numeric output conversion. You can use the words:

```
<#  #  #s  hold  sign  #>
```

to specify the conversion of a 32-bit number into an ASCII character string stored in right-to-left order. See `(.)` and `(u.)` for example usages.

<<

stack: (x1 u -- x2)
code: none
generates: `lshift`

x2 is the result of logically left shifting *x1* by *u* places. Zeroes are shifted into the least-significant bits.

For example:

```
: bljoin  (  byte.low byte.lowmid byte.highmid byte.high -- l )
   8 << +  8 << +  8 << +
;
```

<=

stack: (n1 n2 -- less_than_or_equal?)
code: 43

less_than_or_equal? is true if *n1* is less than or equal to *n2*. *n1* and *n2* are signed integers.

<>

stack: (x1 x2 -- not_equal?)
code: 3D

not_equal? is true if *x1* is not equal to *x2*. *x1* and *x2* are signed integers.

=

stack: (x1 x2 -- equal?)
code: 3C

equal? is true if *x1* is equal to *x2*. *x1* and *x2* are signed integers.

>

stack: (n1 n2 -- greater_than?)
code: 3B

greater_than? is true if *n1* is greater than *n2*. *n1* and *n2* are signed integers.

>=

stack: (n1 n2 -- greater_than_or_equal?)
code: 42

greater_than_or_equal? is true if *n1* is greater than or equal to *n2*. *n1* and *n2* are signed integers.

>>

stack: (x1 u -- x2)
code: none
generates: `rshift`

x2 is the result of logically right shifting *x1* by *u* places. Zeroes are shifted into the most-significant bits. Use `>>a` for signed shifting.

For example:

```
: wbsplit  ( w -- b.low b.high )
   dup  h# ff and  swap   8 >>
   h# ff and
;
```

? "fetch print"

stack: (a-addr --)
code: none
generates: `@ .`

Fetches and prints the 32-bit value at the given address. A standard Forth word, primarily used interactively.

@ "fetch"

stack: (a-addr -- x)
code: 6D

x is the value stored at *a-addr*. For more portable code, use `l@` if you explicitly want a 32-bit access. *a-addr* must be aligned as given by `variable`.

See also: `c@`, `w@`, `l@`, `rb@`, `rw@`, `rl@`

[

stack: (--)
code: none

Enter interpretation state.

['] "bracket tick bracket"

stack: ([old-name< >] -- xt)
code: none
generates: `b(') old-FCode#`

`'` or `[']` is used to generate the execution token (*xt*) of the word immediately following the `'` or `[']`.

`'` should only be used *outside* definitions; `[']` may be used either inside or outside definitions. Examples shown usually use `[']`, since it will always generate the intended result:

```
: my-probe … ['] my-install is-install … ;
```

or

```
['] my-install is-install
```

In normal Forth, `'` may be used within definitions for the creation of language extensions, but such usage is not applicable to FCode Programs.

\

stack: ([rest-of-line<eol>] --)
code: none

Causes the compiler/interpreter to ignore the rest of the input line after the `\` . `\` can occur anywhere on an input line. Note that a space must be present after `\` .

For example:

```
0 value his-ihandle  \ Place to save someone's ihandle
```

See also: `(`, `(s`

]

stack: (--)
code: none

Enter compilation state.

0

stack: (-- 0)
code: A5

Leaves the value 0 on the stack. The only numbers that are not encoded using `b(lit)` are the values -1, 0, 1, 2, or 3. Because these numbers occur so frequently, they are assigned individual FCodes to save space.

0<

stack: (n -- less_than_0?)
code: 36

less_than_0? is true if *n* is less than zero (negative).

0<=

stack: (n -- less_than_or_equal_to_0?)
code: 37

less_than_or_equal_to_0? is true if *n* is less than or equal to zero.

0<>

stack: (n -- not_equal_to_0?)
code: 35

not_equal_to_0? is true if *n* is not zero.

0=

stack: (nu / flag -- equal_to_0?)
code: 34

equal_to_0? is true if *n*u / flag is zero. This word will invert any flag.

0>

stack: (n -- greater_than_0?)
code: 38

greater_than_0? is true if `n` is greater than zero.

0>=

stack: (n -- greater_than_or_equal_to_0?)
code: 39

greater_than_or_equal_to_0? is true if *n* is greater than or equal to zero.

1

stack: (-- 1)
code: A6

Leaves the value 1 on the stack. The only numbers that are not encoded using `b(lit)` are the values -1, 0, 1, 2, or 3. Because these numbers occur so frequently, these values are assigned individual FCodes to save space.

1+

stack: (nu1 -- nu2)
code: none
generates: `1 +`

nu2 is the result of adding 1 to *nu1*.

1-

stack: (nu1 -- nu2)
code: none
generates: `1 -`

nu2 is the result of subtracting 1 from *nu1*.

-1

stack: (-- -1)
code: A4

Leaves the value -1 on the stack. The only numbers that are not encoded using `b(lit)` are the values -1, 0, 1, 2, or 3. Because these numbers occur so frequently, these values are assigned individual FCodes to reduce the resulting FCode image size.

2

stack: (-- 2)
code: A7

Leaves the value 2 on the stack. The only numbers that are not encoded using `b(lit)` are the values -1, 0, 1, 2, or 3. Because these numbers occur so frequently, these values are assigned individual FCodes to reduce the resulting FCode image size.

2! “two store”

stack: (x1 x2 a-addr --)
code: 77

x1 and *x2* are stored in consecutive 32-bit locations starting at *a-addr. x2* is stored at the lower address. This is equivalent to: `swap over ! cell+ !`.

2*

stack: (x1 -- x2)
code: 59

x2 is the result of shifting *x1* left one bit. A zero is shifted into the vacated bit position. This is equivalent to multiplying by 2.

2+

stack: (nu1 -- nu2)
code: none
generates: `2 +`

nu2 is the result of adding 2 to *nu1*.

2-

stack: (nu1 -- nu2)
code: none
generates: `2 -`

nu2 is the result of subtracting 2 from *nu1*.

2/ “two slash”

stack: (x1 -- x2)
code: 57

x2 is the result of arithmetically shifting *x1* right one bit. The sign is included in the shift and remains unchanged. This is equivalent to dividing by 2.

2@ “two fetch”

stack: (a-addr -- x1 x2)
code: 76

x1 and *x2* are two numbers stored in consecutive 32-bit locations starting at *a-addr. x2* is the number that was stored at the lower address. This is equivalent to: `dup cell+ @ swap @`.

3

stack: (-- 3)
code: A8

Leaves the value 3 on the stack. The only numbers that are not encoded using `b(lit)` are the values -1, 0, 1, 2, or 3. Because these numbers occur so frequently, these values are assigned individual FCodes to reduce the resulting FCode image size.

>>a

stack: (x1 u -- x2)
code: 29

x2 is the result of arithmetically right shifting *x1* by *u* places. The sign bit of *x1* is shifted into the most-significant bits (i.e. sign extend the high bit).

For example:

```
ok ffff.0000 6 >>a .h
fffffc00
ok ffff.0000 6 >> .h
3fffc00
```

abort

stack: (… --) (R: … --)
code: 216

Aborts program execution, clearing the data and return stacks. Control returns to the `ok` prompt. Called after encountering fatal errors.

For example:

```
: probe-loop  ( addr -- )
   begin dup l@ drop key? if  abort  then  again
   \ generate a tight probe loop until any key is pressed.
;
```

See also: `exit`

abort" “abort quote”

stack: (C: [text<">] --)
(… abort? -- … | <nothing>) (R: … -- … | <nothing>)
code: none

If *abort*? is non-zero, display *text* and call `abort`. Leading spaces in *text* are *not* ignored and end-of-line is *not* treated as a delimiting space.

Although `abort"` is not available as an FCode, the same affect can be achieved with a phrase like

```
if ." error-text" -2 throw then
```

abs

stack: (n -- u)
code: 2D

u is the absolute value of *n*. If *n* is the maximum negative number, *u* is the same value since the maximum negative number in two's complement notation has no positive equivalent.

accept

stack: (addr len1 -- len2)
code: none
generates: `span @ -rot expect span @ swap span !`

Get an edited input line, storing it at *addr*. len1 is maximum allowed length. len2 is actual length received.

For example:

```
h# 10 buffer: my-name-buff
: hello ( -- )
   ." Enter Your First name " my-name-buff h# 10 accept
   ." FirmWorks Welcomes " my-name-buff swap type cr
;
```

### "address"

The standard property name which specifies the virtual addresses of one or more memory-mapped regions of the associated device. This property is typically used to report the virtual addresses of regions that the firmware has already mapped such that client programs can re-use those mappings.

The `"address"` property should be created after a virtual address has been mapped and should delete the `"address"` property when that mapping is destroyed.

See also: `free-virtual`

"address-bits"

The standard property name for use with `"network"` devices which indicates the number of bits needed to address this device on the physical layer of the network. The absence of this property implies the default value of 48.

### "#address-cells"

The standard property name used with packages that define a physical address space i.e. those packages with a `"decode-unit"` method. This property specifies the number of cells that are used to encode a physical address within that package's address space. The absence of this property in a package with a `"decode-unit"` method implies a default value for this property of 2.

See also: `map-in`, `map-low`, `decode-unit`, `my-address`, `my-space`, `my-unit`, `encode-phys`, and `decode-phys`.

.adr

stack: (addr --)
code: none

Displays in symbolic form the symbol associated with the address nearest to (but not greater than) *addr.* The symbolic form of an address is usually a symbol name plus a non-negative numeric offset.

See also: `value>sym`

again

stack: (C: dest-sys --)
(--)
code: none
generates: `bbranch -offset`

Used in the form `begin`…`again` to generate an infinite loop. Use a keyboard abort, or `abort` or `exit`, to terminate such a loop. Use this word with caution!

For example:

```
: probe-loop  ( addr -- )
   \ generate a tight probe loop until any key is pressed.
   begin dup l@ drop key? if  abort  then  again
;
```

See also: `repeat`, `until`, `while`

alarm

stack: (xt n --)
code: 213

Arranges to execute the package method *xt* at periodic intervals of *n* milliseconds (to the best accuracy possible). If *n* is 0, stop the periodic execution of *xt* within the current instance context (leaving unaffected any periodic execution of *xt* that was established within a different instance).

xt is the execution token, as returned by `[']`. *xt* must be the execution token of a method which neither expects stack arguments nor leaves stack results i.e. whose stack diagram is `( -- )`.

`alarm` executes in the context in which it was installed. Each time the method is called, the current instance will be set to the same as the current instance at the time that `alarm` was executed and the current instance will then be restored to its previous value afterwards. `alarm` must be removed prior to closing the instance which installed it.

A common use of `alarm` would be to implement a console input device's polling function.

For example:

```
: my-checker  ( -- )  test-dev-status  if  user-abort  then  ;
: install-abort  ( -- )  ['] my-checker d# 10 alarm  ;
```

alias

stack: ("new-name< >old-name< >" --)
code: none

`alias` creates a new name, with the exact behavior of some other existing name. The new name can then be used interchangeably with the old name and have the same effect.

The tokenizer does *not* generate any FCode for an `alias` command, but instead simply updates its own lookup table of existing words. Any occurrence of *new-name* causes the assigned FCode value of *old-name* to be generated. One implication is that *new-name* will not appear in the Open Firmware dictionary after the FCode Program is compiled. If this behavior is undesirable, use a colon definition instead.

If the original FCode source text is downloaded and interpreted directly, without being tokenized or detokenized, then any new `alias` words *will* show up and be usable directly.

For example:

```
alias pkg-prop get-package-property
```

"/aliases"

The standard node containing this system's device alias list. The value of the `name` property of this node is "`aliases`". The remaining properties of this node constitute the device alias list. For each such property, the property name is the name of an alias and the property value is the alias's expansion encoded with `encode-string`.

align

stack: (--)
code: none

Allocates dictionary bytes as necessary to leave the top of the dictionary variable aligned.

aligned

stack: (n1 -- n1 | a-addr)
code: AE

Increases *n1* as necessary to yield a variable aligned address. If *n1* is already aligned, returns *n*1. Otherwise, returns the next higher variable aligned address, *a-addr.*

alloc-mem

stack: (len -- a-addr)
code: 8B

Allocates a buffer of *len* of physical memory that has been aligned to the most stringent requirements of the processor. If successful, returns the associated virtual address. If not successful, `throw` will be called with an appropriate error message as with `abort"`. Memory allocated by `alloc-mem` is not suitable for DMA.

To detect an out-of-memory condition:

```
h# 100 ['] alloc-mem catch ?dup if
   throw
else
   ( virt ) constant my-buff
then
```

See also: `abort"`, `dma-alloc`, `free-mem`, `throw`.

allot

stack: (len --)
code: none
generates: `0 max 0 ?do 0 c, loop`

Allocates *len* bytes in the dictionary. If the operation fails, a `throw` will be called with an appropriate error message as with `abort"`. Error conditions can be detected and handled properly with the phrase `[']` `allot` `catch`.

"alternate-reg"

This property describes alternative access paths for the addressable regions described by the `"reg"` property. Typically, an alternative access path exists when a particular part of a device can be accessed either in memory space or in I/O space, with a separate base address register for each of the two access paths. The primary access paths are described by the `"reg"` property and the secondary access paths, if any, are described by the `"alternate-reg"` property.

If no alternative paths exist, the `"alternate-reg"` property should not be defined. If the device has alternative access paths, each entry (i.e. each *phys-addr size* pair) of its value represents the secondary access path for the addressable region whose primary access path is in the corresponding entry of the `"reg"` property value. If the number of `"alternate-reg"` entries exceeds the number of `"reg"` property entries, the additional entries denote addressable regions that are not represented by `"reg"` property entries, and are thus not intended to be used in normal operation; such regions might, for example, be used for diagnostic functions. If the number of `"alternate-reg"` entries is less than the number of `"reg"` entries, the regions described by the extra `"reg"` entries do not have alternative access paths. An `"alternate-reg"` entry whose *phys.hi* component is zero indicates that the corresponding region does not have an alternative access path; such an entry can be used as a “place holder” to preserve the positions of later entries relative to the corresponding `"reg"` entries. The first `"alternate-reg"` entry, corresponding to the `"reg"` entry describing the function’s Configuration Space registers, has a *phys.hi* component of zero.

The property value is an arbitrary number of (*phys-addr, size*) pairs where:

- *phys-addr* is (*phys.lo phys.mid phys.hi*), encoded with `encode-phys`.
- *size* is a pair of integers, each encoded with `encode-int`. The first integer denotes the most-significant 32 bits of the 64-bit region size and the second integer denotes the least-significant 32 bits thereof.

See also: `"reg"`

and

stack: (x1 x2 -- x3)
code: 23

x3 is the bit-by-bit logical and of *x1* with *x2*.

apply

stack: (… "method-name< >device-specifier< >" -- ???)
code: none

Executes the named method in the specified package by performing the function of `execute-device-method`. If the operation fails, a `throw` will be called with an appropriate error message as with `abort"`. Error conditions can be detected and handled properly with the phrase `['] apply catch`.

ascii

stack: ([text< >] -- char)
code: none
generates: `b(lit) 00 00 00 value`

Skips leading space delimiters and puts the ASCII value of the first letter in *text* on the stack. For example:

```
ascii C ( equals hex 43 )
ascii c ( equals hex 63 )
```

`ascii` may be used either inside or outside of definitions. `ascii` is equivalent to `[char]`, but `[char]` is ANS standard Forth.

See also: `char`, `[char]`

assign-addresses

stack: (--)
code: none

This User Interface word is intended to be used for debugging FCode within the context of `begin-package`…`end-package`. Executing this word causes addresses to be assined to this node creating an `"assigned-addresses"` property reflecting those addresses. This word simulates the action of the FCode probing process for PCI devices, and should be executed after evaluating the FCode for the node and before the execution of `end-package.`

auto-boot?

stack: (-- auto?)
code: none

If the configuration variable `auto-boot?` is set to true after power-on or `reset-all`, the command string specified by `boot-command` will be executed. In the usual case, the value of `boot-command` is `boot`. Usually, `boot` transfers control to a client program.

If `auto-boot?` is set to false, the User Interface command interpreter is entered.

"available"

This property defines resources that are managed by this package that are currently available for use by a client program. The `claim` and `release` methods affect the value of this property.

The property value is an arbitrary number of (*phys-addr*, *length*) pairs where:

- *phys-addr* is a phys.lo phys.mid phys.hi list of integers encoded with `encode-int`.
- *length* (whose format depends on the package) is one or more integers, each encoded with `encode-int`.

See also: `claim`, `"existing"`, `"reg"`, `release`

b(")

stack: (-- str len)
code: 12

An internal word, generated by `"`, `."` and `.(` which leaves a text string on the stack. Never use the word `b(")` in source code.

b(')

stack: (-- xt)
code: 11

An internal word, generated by `'` and `[']` which leaves the execution token of the immediately following word on the stack. The FCode for `b(')` should always be followed by the FCode of the desired word. Never use the word `b(')` in source code.

b(:)

stack: (--)
code: B7

An internal word generated by the defining word `:` . Never use the word `b(:)` in source code.

b(;)

stack: (--)
code: C2

An internal word generated by `;` to end a colon definition. Never use the word `b(;)` in source code.

banner

stack: (--)
code: none

Displays the system power-on banner in a system-dependent screen location (usually at the top of the screen or at the current cursor position).

If the current output device has a `"device_type"` property whose value is `"display"`, display a logo by executing the current output device's `draw-logo` method with the following arguments:

- The `line#` argument is either 0 or the line number corresponding to the current cursor position, at the system's discretion.
- If `oem_logo?` is true, the `addr` argument is the address returned by `oem-logo`. Otherwise, it is the address of the system-dependent default logo.
- The `width` and `height` arguments are both 64.

In any case:

- If `oem-banner?` is true, display the text given by the value of `oem-banner`.
- Otherwise, display implementation-dependent information about the system, for example the machine type, serial number, firmware revision, network address, and hardware configuration.

If `banner` is executed from within the NVRAM script, suppress automatic execution of the following Open Firmware start-up sequence:

- `probe-all`
- `install-console`
- `banner`

For a usage example, see "Patching FCode of a Plug-in Card" on page 22.

See also: `suppress-banner`

base

stack:	(-- a-addr)
code:	A0

base is the `variable` that contains the current numeric conversion radix to be used when the FCode Program is executing, such as 10 for decimal, 16 for hex, 8 for octal, and so on. Like any variable, `base` leaves its address on the stack.

For example, to print the current value of `base`, use:

```
base @ .d
```

The tokenizer words `decimal`, `hex`, or `octal` are also available for changing the value in `base` as desired. However, these four words behave differently depending whether they occur within a definition or outside of a definition.

If any of `decimal`, `hex`, or `octal` occur *within* a definition, then they will be compiled, later causing a change to the value in `base` when that definition is executed. This can be a useful affect when, for example, a device's `open` method must interpret a number in an argument string. In such a case, however, the value in `base` should be saved prior to changing the base, and the old value of `base` should be restored prior to leaving the definition in which the number base was changed. Failure to do this can be very confusing to the user who will have caused the number base of the machine to change without explicitly attempting to do so.

If any of `decimal`, `hex`, or `octal` occur *outside* of a definition, however, then they are interpreted as commands to the tokenizer program itself, thus affecting the interpretation of all subsequent numbers in the text.

Note that changes to `base` affect the numeric base of the User Interface, which can create much confusion for any user (the default value for `base` is hexadecimal). If you *must* change the base, it is recommended that you save and then restore the original base, as in:

```
: .o ( n -- )  \ Print n in octal
  base @ swap    ( oldbase n )
  octal .        ( oldbase )
  base !
;
```

In general, only numeric *output* will be affected by the value in `base`. Fixed numbers in FCode source are interpreted by the tokenizer program. Most numeric input is controlled by `decimal`, `hex`, `octal`, `d#`, `h#`, and `o#`, but these words only affect the tokenizer input base; they but do *not* affect the value in `base`.

For example:

```
              (Assume the initial value in base is 16, i.e. User Interface is in hex)
               (No assumptions should be made about the initial tokenizer base)
version1
hex            (Tokenizer in base 16; later execution, using base, in base 16)
20 .           (Compile decimal 32, later print "20" when FCode executes)
decimal        (Tokenizer is in base 10, later execution is in base 16)
20 .           (Compile decimal 20, later print "14" since FCode executes in hex)
: TEST ( -- )
    octal      (Still compiling in decimal, later change base when TEST executes)
    20 .       (Compiles decimal 20, prints "24" since base was just changed)
    h# 20 .d  (Compiles decimal 32, prints "32"; no permanent base changes)
    20 .       (Compiles decimal 20, prints "24")
;
20 .           (Compile decimal 20, later print "14"
TEST           (Prints "24 32 24"; has a side-effect of changing the base)
20 .           (Compile decimal 20, later print 24 since TEST changed base)
hex           (Tokenizer is in base 16; later execution, using base, still in base 8)
20 .           (Compile decimal 32, later print "40")
```

If this all seems confusing, simply follow these guidelines:

Good: Initially declare `hex` just after `fcode-version2`, and make liberal use of `d#`, `o#`, `h#`, `.h` and `.d`.

Bad: Changing `base` within a definition (either directly or by calling `decimal`, `hex`, or `octal`) without restoring the previous base before reaching the end of the definition.

bbranch

stack: (--)
code: 13

An internal word generated by `again`, `repeat`, and `else` which causes an unconditional branch. Never use the word `bbranch` in source code.

b?branch

stack: (flag --)
code: 14

An internal word generated by `until`, `while`, and `if` which causes a conditional branch. Never use the word `b?branch` in source code.

b(buffer:)

stack: (n --)
code: BD

An internal word generated by the defining word `buffer:` which allocates *n* bytes of storage space. Never use the word `b(buffer:)` in source code.

b(case)

stack: (sel-- sel)
code: C4

An internal word generated by `case`. Never use the word `b(case)` in source code.

b(constant)

stack: (n --)
code: BA

An internal word generated by the defining word `constant`. Never use the word `b(constant)` in source code.

b(create)

stack: (--)
code: BB

An internal word generated by the defining word `create`. Never use the word `b(create)` in source code.

b(defer)

stack: (--)
code: BC

An internal word generated by the defining word `defer`. Never use the word `b(defer)` in source code.

b(do)

stack: (end start --)
code: 17

An internal word generated by `do`. Never use the word `b(do)` in source code.

b(?do)

stack: (end start --)
code: 18

An internal word generated by `?do`. Never use the word `b(?do)` in source code.

begin

stack: (C: -- dest-sys)
(--)
code: none
generates: `b(<mark)`

Marks the beginning of a conditional loop, such as `begin`…`until`, `begin`…`while`…`repeat`, or `begin`…`again`. See these other words for more details.

begin-package

stack: (arg-str arg-len reg-str reg-len dev-str dev-len --)
code: none

Prepares to create a new node in the device tree by performing the following operations:

- The parent device (and all higher parents) is opened with `open-dev` using the parameters *dev-str* and *dev-len*. If `open-dev` is unsuccessful, execution is terminated with an error message.
- `my-self` is set to the new parent ihandle.
- The active package is set to the parent device.
- The child node is opened with `new-device`.
- The child arguments contained in the parameters *arg-str*, *arg-len*, *reg-str* and *reg-len* are set with `set-args`.

dev-str and *dev-len* contain the device-path string of the parent of the child about to be created. The form of the device-path string is either a full device pathname or a pre-existing device alias.

reg-str and *reg-len* contain the unit-address string, the text representation of the physical address of the child within the address space of the parent. For PCI, an example would be `"3,0"`. (The child can retrieve the numerical form of the unit-address with `my-address` and `my-space`.)

arg-str and *arg-len* contain the instance-arguments string. (The child can retrieve this value with `my-args`.)

For example:

```
0 0 " 3,0" " /pci" begin-package
```

Note – Since Open Firmware is only required to provide two buffers for the interpreter's use in assembling strings, trying to pass an argument string, a unit address string and a device-path string directly to `begin-package` through the interpreter is likely to fail.

A simple work-around is to define a word containing, say, the argument string and so use the compiler to assemble the string and hold it in the dictionary. This word would then be called to push the argument string onto the stack.

For example:

```
: arg$ " my begin package args" ;
arg$ " 0,0,0" " /pci" begin-package
```

begin-select

stack: ("device-specifier< >" --)
code: none

A User Interface extension provided by some implementations (e.g. FirmWorks / Sun).

Creates an instance chain for the device specified by *device-specifier* except that the `open` method of the leaf node is *not* called. `begin-select` is useful for debugging the `open` method of a driver by allowing the `open` method to be called under the control of the debugger. For example:

```
ok begin-select foo
ok debug open open
```

See also: “Using begin-select” on page 38.

begin-select-dev

stack: (dev-str dev-len --)
code: none

A User Interface extension provided by some implementations (e.g. FirmWorks / Sun).

Creates an instance chain for the device specified by *dev-str dev-len* except that the `open` method of the leaf node is *not* called. `begin-select`-dev is useful for debugging the `open` method of a driver by allowing the `open` method to be called under the control of the debugger. For example:

```
ok " foo" begin-select-dev
ok debug open open
```

See also: “Using begin-select-dev” on page 38.

behavior

stack: (defer-xt -- contents-xt)
code: DE

This command is used to retrieve the execution contents of a defer word.

A typical use would be to fetch and save the current execution of a defer word, change the behavior temporarily and later restore the original behavior. For example:

```
defer my-func
0 value old-func
['] framus is my-func …
['] my-func behavior is old-func
['] foo is my-func
… my-func …
old-func is my-func
```

bell

stack: (-- 0x07)
code: AB

Leave the ASCII code for the bell character on the stack.

b(endcase)

stack: (sel --)
code: C5

An internal word generated by `endcase`. Never use the word `b(endcase)` in source code.

b(endof)

stack: (--)
code: C6

An internal word generated by `endof`. Never use the word `b(endof)` in source code.

between

stack: (n min max -- min<=n<=max?)
code: 44

min<=n<=max? is true if *n* is between *min* and *max*, inclusive of both endpoints.

See `within` for a different form of comparison.

b(field)

stack: (addr -- addr+offset)
code: BE

An internal word generated by the defining word `field`. Never use the word `b(field)` in source code.

bl “bee el”

stack: (-- 0x20)
code: A9

Leaves the ASCII code for the space character on the stack.

blank

stack: (addr len --)
code: none
generates: `bl fill`

Sets *len* bytes of memory beginning at *addr* to the ASCII character value for space (hex 20). No action is taken if *len* is zero.

b(leave)

stack: (--)
code: 1B

An internal word generated by `leave`. Never use the word `b(leave)` in source code.

blink-screen

stack: (--)
code: 15B

A `defer` word, called by the terminal emulator, when it has processed a character sequence that calls for ringing the console bell, but the console input device package has no `ring-bell` method.

`blink-screen` is initially empty, but *must* be loaded with a system-dependent routine in order for the terminal emulator to function correctly. The routine must cause some momentary discernible effect that leaves the screen in the same state as before.

This can be done with `to`, or it can be loaded automatically with `fb1-install` or `fb8-install` (which load `fb1-blink-screen` or `fb8-blink-screen`, respectively). These default routines invert the screen (twice) by xor-ing every visible pixel. This is quite slow.

A replacement routine simply disables the video for 20 milliseconds or so, i.e.

```
: my-blink-screen  ( -- )   video-off  20 ms  video-on  ;
…
   \ load default behaviors with fbx-install, then:
   ['] my-blink-screen  to blink-screen
```

Of course, this example assumes that your display hardware is able to quickly enable and disable the video without otherwise affecting the state.

b(lit)

stack: (-- n)
code: 10

An internal word used to save numbers. Never use the word `b(lit)` in source code.

The only numbers that are not encoded using `b(lit)` are the values -1, 0, 1, 2, or 3. Because these numbers occur so frequently, these values are assigned individual FCodes to save space.

bljoin

stack: (byte.lo byte2 byte3 byte.hi -- quad)
code: 7F

Merges the least significant byte from each of the four input stack items into a single 32-bit word. All other bits of the input stack items must be zero to guarantee correct results.

"block"

This is the standard property value of the `"device_type"` property for random access, fixed-length block storage devices (i.e. hard and floppy disks, CDROMs).

Although devices of type `"block"` present a byte-oriented interface to the rest of the system, the associated hardware devices are usually block-oriented [i.e. the device reads and writes data in “blocks” (groups of, for example, 512 or 2048 bytes)]. The standard `deblocker` support package assists in the presentation of a byte-oriented interface above an underlying block-oriented interface, implementing a layer of buffering that “hides” the underlying block length.

`"block"` devices are often subdivided into several logical “partitions” as defined by a “disk label” - a special block, usually the first one, containing information about the device. The driver is responsible for appropriately interpreting a disk label. The driver may use the standard `disk-label` support package if it does not implement a specialized label. The `disk-label` support package interprets a system-dependent label format. Since the disk booting protocol usually depends upon the label format, the standard `disk-label` support package also implements a load method for the corresponding boot protocol.

Devices of type `"block"` must implement the following methods:

- `open`
- `close`
- `read`
- `seek`
- `load`

If the device is writable, the `write` method should also be implemented.

`block` devices often use the `deblocker` support package to implement the `read`, `write`, and `seek` methods, and the `disk-label` support package to implement the `load` method.

block-size

stack:	(-- block-len)
code:	none

`block-size` returns the “granularity” in bytes for accesses to this device. All transfers to the device should be multiples of this size.

If *block-len* is 1, the device supports arbitrary transfer sizes up to the value specified by `max-transfer`.

b(loop)

stack:	(--)
code:	15

An internal word generated by `loop`. Never use the word `b(loop)` in source code.

b(+loop)

stack:	(n --)
code:	16

An internal word generated by `+loop`. Never use the word `b(+loop)` in source code.

b(<mark)

stack:	(--)
code:	B1

An internal word generated by `begin`. Never use the word `b(<mark)` in source code.

body> “body from”

stack: (a-addr -- xt)
code: 85

Converts the data field address of a word to its execution token.

>body “to body”

stack: (xt -- a-addr)
code: 86

Converts the execution token of a word to its data field address.

b(of)

stack: (testval --)
code: 1C

An internal word generated by `of`. Never use the word `b(of)` in source code.

boot

stack: ("{param-text}<eol>" --)
code: none

Loads and executes the program specified by *param-text* by:

- Ensuring a suitable state for booting,
- Performing the function of `load` to load a client program from the device (if any) specified by *param-text*, and
- If `load` succeeds, perform the function of `go` to execute the client program.

For example:

```
ok boot
ok boot device-specifier
ok boot arguments
ok boot device-arguments
```

boot-command

stack: (-- addr len)

The value of this configuration variable is a string describing the boot command to be used if `auto-boot?` is true.

The suggested default value is “boot”.

boot-device

stack: (-- dev-str dev-len)

The value of this configuration variable is a string describing the device name and any required filename to be used by `boot` if `diagnostic-mode?` is false. The string is a device specifier or a list of device specifiers as described in the definition of `load`.

The suggested default value is “disk”.

boot-file

stack: (-- arg-str arg-len)

The value of this configuration variable is a string describing the default arguments to be used by `boot` if `diagnostic-mode?` is false.

The suggested default value is the empty string.

"bootargs"

This property appears in the `/chosen` node if a `boot` or a `load` command has been issued since Open Firmware was last reset. The property value is the arguments field of the most recent `boot` command.

"bootpath"

This property appears in the `/chosen` node if a `boot` or a `load` command has been issued since Open Firmware was last reset. The property value is the complete device path to which the device specifier of that last command was resolved.

bounds

stack: (start cnt -- start+cnt start)
code: AC

Converts a starting value and count into the form required for a `do` or `?do` loop. For example, to perform a loop 20 times, counting up from `4000` to `401f` inclusive, use:

```
4000 20 bounds do…loop
```

This is equivalent to:

```
4020 4000 do…loop
```

+bp

stack: (addr --)
code: none

Adds the given address to the breakpoint list.

-bp

stack: (addr --)
code: none

Removes the breakpoint at the given address.

--bp

stack: (--)
code: none

Removes the most recently set breakpoint.

.bp

stack: (--)
code: none

Displays a list of all of the addresses at which breakpoints are set.

bpoff

stack: (--)
code: none

Removes all breakpoints.

.breakpoint

stack: (--)
code: none

`.breakpoint` is a `defer` word that is executed whenever a breakpoint occurs. The default behavior is `.instruction`.

See also: `defer`

b(>resolve)

stack: (--)
code: B2

An internal word generated by `repeat` and `then`. Never use the word `b(>resolve)` in source code.

bs

stack: (-- 0x08)
code: AA

Leaves the ASCII code for the backspace character on the stack.

b(to)

stack: (--)
code: C3

An internal word generated by `to`. Never use the word `b(to)` in source code.

buffer:

stack: (len "new-name< >" --)(E: -- a-addr)
code: none
generates: `new-token|named-token|external-token b(buffer:)`

Allocates *len* bytes of storage space and creates a name, *new-name*. When *new-name* is executed, it leaves the address of the first byte of the buffer on the stack.

For example:

```
200 buffer: name
name ( addr )
```

b(value)

stack: (n --)
code: B8

An internal word generated by the defining word `value`. Never use the word `b(value)` in source code.

b(variable)

stack: (n --)
code: B9

An internal word generated by the defining word `variable`. Never use the word `b(variable)` in source code.

bwjoin

stack: (byte.lo byte.hi -- w)
code: B0

Merges the least significant byte of each of the two input stack arguments into a doublet. Correct results are only guaranteed if all other bits of the input stack arguments are zero.

bxjoin

stack: (b.lo b.2 b.3 b.4 b.5 b.6 b.7 b.hi -- o)
code: 241

Join 8 bytes to form an octlet. The high-order bits of each of the bytes are ignored.

This function is only available on 64-bit implementations.

"byte"

This is the standard property value of the `"device_type"` property for sequential access, record-oriented storage devices (e.g. tape).

Although devices of type "byte" present a byte-oriented interface to the rest of the system, the associated hardware devices are usually record-oriented (i.e. the device reads and writes data in "records" containing more than one byte). The records may be either fixed length (all records must be the same length) or variable length (the record length may vary from record to record). Tapes may be subdivided into several tape files delimited by file marks.

The standard `deblocker` support package assists in the presentation of a byte-oriented interface above an underlying record-oriented interface, implementing a layer of buffering that “hides” the underlying record structure.

Devices of type `"byte"` must implement the following methods:

- `open`
- `close`
- `read`
- `seek`

  The `seek` method locates the byte numbered *pos.lo* with the tape file *pos.hi*. If *pos.lo* and *pos.hi* are both zero, the tape is rewound. `seek` returns *false* if successful and *true* if unsuccessful.

- `load`

  The `load` method reads a client program from the tape file specified by the value of the *instance-argument* text string (as returned by `my-args`). That value is the string representation of a decimal integer. If the *instance-argument* string is empty, tape file 0 is used. The file read is placed into memory at *addr*, returning *len*, the number of bytes actually read.

If the device is writable, the `write` method should also be implemented.

`byte` devices often use the `deblocker` support package to implement the `read`, `write`, and `seek` methods.

byte-load

stack: (addr xt --)
code: 23E

Interprets the FCode Program located at `addr`. If `xt` is 1, use `rb@` to read the FCode Program, otherwise use `xt` as the execution token of the definition to be used to read the FCode Program. Continue reading and interpreting the program until `end0` is encountered.

Be aware that `byte-load` does not itself create a new device node as a "container" for any properties and methods defined by the FCode Program that `byte-load` evaluates. If the FCode Program is intended to create such a node, appropriate preparation must be done before and after executing `byte-load`. For example, `new-device` and `set-args` can be executed before and `finish-device` can be executed after `byte-load` is executed.

If `byte-load` is to be executed from the User Interface, additional set up is usually necessary before executing `new-device`; see `begin-package` for more details.

c! "see store"

stack: (byte addr --)
code: 75

Stores the least significant 8 bits of *byte* in the byte at *addr*.

See also: `rb!`

c, "see comma"

stack: (byte --)
code: D0

Compiles a byte into the dictionary. `c,` can be used, in conjunction with `create`, to create an array-type structure, as:

```
create yellow  77 c, 23 c, ff c, ff c, 47 c, 22 c, …
```

Later execution of *yellow* leaves the address of the first byte of the array (i.e. the address of the byte "77") on the stack.

c; “see semicolon”

stack: (C: code-sys --)
(--) (R: -- nest-sys)
code: none

Ends the creation of a machine code word by assembling code that will, upon execution, return control to the calling routine specified by *nest-sys*.

/c “per see”

stack: (-- n)
code: 5A

Leaves the number of address units to a byte (i.e. 1) on the stack.

See also: `/w`, `/l`, `/n`

/c* “per see star”

stack: (nu1 -- nu2)
code: none
generates: `chars`

Synonym for `chars`.

c@ “see fetch”

stack: (addr --byte)
code: 71

Fetches the byte at address *addr* and leaves it on top of the stack with the high order bytes filled with zeroes.

See also: `rb@`

ca+ “see ay plus”

stack: (addr1 index -- addr2)
code: 5E

Increments *addr1* by *index* times the value of `/c`. `ca+` should be used in preference to `+` when calculating addresses because it more clearly expresses the intent of the operation and is more portable.

ca1+ “see ay one plus”

stack: (addr1 -- addr2)
code: none
generates: `char+`

Synonym for `char+`

callback

stack: ("service-name< >" "arguments<eol>" --)
code: none

Executes the specified client program callback routine.

$callback

stack: (argn … arg1 nargs addr len -- retn … ret2 Nreturns-1)
code: none

Executes the specified client program callback routine.

$call-method

stack: (… method-str method-len ihandle -- ???)
code: 20E

Executes the device interface method *method-str method-len* within the open package instance *ihandle*. The ellipses (…) indicate that the contents of the stack before and after the method is called depend upon the particular method being called.

For example:

```
: dma-alloc  ( #bytes -- virt )  " dma-alloc"  my-parent $call-method  ;
```

See also: `open-package`.

call-package

stack: (… xt ihandle -- ???)
code: 208

Executes the device interface method *xt* within the open package instance *ihandle*. The ellipses (…) indicate that the contents of the stack before and after the method is called depend upon the particular method being called.

For example:

```
0 value label-ihandle  \ place to save the ihandle of other package
0 value offset-method  \ place to save the xt of found method
: init ( -- )
   my-args " disk-label" $open-package  ( ihandle )
   to label-ihandle
   " offset" label-ihandle
   ihandle>phandle ( name-addr name-len phandle )
   find-method if  ( xt )
      to offset-method
   else  (    )
      ." Can't find offset method "
   then
;
init
: add-offset ( d.byte# -- d.bytes# )
   offset-method label-ihandle call-package
;
```

See also: `find-method`, `open-package`

$call-parent

stack: (… method-str method-len -- ???)
code: 209

Calls the method named by *method-str method-len* within the parent instance. If the called package has no such method, an error is signaled with `throw`. Equivalent to:

```
my-parent $call-method
```

The ellipses (...) and question marks (???) indicate that the contents of the stack before and after the method is called depend upon the particular method being called.

For example, since the stack diagram for `dma-alloc` is (size -- virt), *size* must be pushed on the stack followed by *method-str* and *method-len* prior to calling `$call-parent` which subsequently returns *virt*:

```
: my-dma-alloc  ( -- virt )h# 2000 " dma-alloc"$call-parent ;
```

.calls

"dot calls"

stack: (xt --)
code: none

Displays a list of all of the commands which directly use the execution token *xt*.

For example, if `framus` calls `foo` and `bar` calls `framus`:

```
['] foo .calls
```

will display `framus` and not `bar`.

carret

stack: (-- 0x0D)
code: none
generates: `b(lit) 00 00 00 0x0D`

Leaves the ASCII code for "carriage return" (i.e. Control-M) on the stack.

case

stack: (selector -- selector)
code: none
generates: `b(case)`

Starts a `case` statement that selects its action based on the value of *selector*. For example:

```
: foo ( selector -- )
  case
    0 of ." It was 0" endof
    5 of ." It was 5" endof
   -2 of ." It was -2" endof
  endcase
;
```

`of` tests the top of the stack against *selector* at run time.

- If they are the same, both the top stack item and *selector* are dropped and the code between `of` and the next `endof` is executed. Program control then continues after the `endcase`.

- If they are not the same, the top stack item is dropped and execution continues at the point just following the matching `endof` with *selector* on the top of the stack.

`endcase` expects an item (typically *selector*) on top of the stack and drops it.

An optional "default clause" may be implemented by placing code between the last `endof` and the `endcase`. When such a default clause is executed, *selector is* on the stack. The default clause may use *selector*, but the default clause must leave an item on the stack for `endcase` to drop. The item left for `endcase` to drop need not be *selector*. For example:

```
: bar ( selector -- value )
  case
    3 of 21 endof
    4 of 33 endof
    1+ 0 \ Default clause. Use selector and push 0 for endcase to drop
  endcase
;
```

`case` statements can be used both inside and outside of colon definitions.

catch

stack: (… xt -- ??? error-code | ??? false)
code: 217

Creates a new error handling context and executes *xt* in that context.

If a `throw` (see below) is called during the execution of *xt*,

1. The error handling context is removed
2. The stack depth is restored to the depth that existed prior to the execution of *xt* (not counting the *xt* stack item)
3. The error code that was passed to `throw` is pushed onto the stack
4. `catch` returns

If `throw` is not called during the execution of *xt*, the error handling context is removed and `catch` returns a `false`. The stack effect is otherwise the same as if *xt* were executed using `execute`.

For example:

```
: add-n-check-limit ( n1 n2 n3  -- n )
   + + dup h# 30  >  if  true throw  then
;
: add-me ( n1 n2 n3  -- a b c | n1+n2+n3 )
    ['] add-n-check-limit catch if
      ." Sum exceeds limit " .s
    else
      ." Sum is within limit. Sum = " .s
    then cr
;
```

Note that, given this definition:

```
1 2 3 add-me
```

shows:

```
Sum is within limit. Sum = 6
```

while:

```
10 20 30 add-me
```

may show something like:

```
Sum exceeds limit  50 9 12
```

Note – Upon a non-zero `throw`, only the stack depth is guaranteed to be the same as before `catch`, not the data stack contents.

cell+

stack: (addr1 -- addr2)
code: 65

Increments *addr1* by the value of `/n`. `cell+` should be used in preference to `wa1+` or `la1+` when the intent is to address items that are the same size as items on the stack.

cells

stack: (nu1 -- nu2)
code: 69

nu2 is the result of multiplying *nu1* by `/n`, the length in bytes of a normal stack item. This is useful for converting an index into a byte offset.

char

stack: ("text< >" -- char)
code: none

Leaves the ASCII code for the next non-whitespace character in the input buffer on the stack. Only for use outside of definitions.

See also: `ascii`, `[char]`

char+

stack: (addr1 -- addr2)
code: 62

Increments *addr1* by the value of `/c`. `char+` should be used in preference to `+` when calculating addresses because it more clearly expresses the intent of the operation and is more portable.

[char]

stack: ([text< >] -- char)
code: none

Leaves the ASCII code for the next non-whitespace character in the input buffer on the stack. For example:

```
[char] C ( equals hex 43 )
[char] c ( equals hex 63 )
```

See also: `ascii`, `char`

"character-set"

stack: This standard property applies to packages implementing `"device_type"` of `"serial"` or `"display"`. The value of this property defines the character set for this device. A typical value is “ISO8859-1”.

See *IEEE Standard 1275-1994* for more details.

char-height

stack: (-- height)
code: 16C

A `value`, containing the standard height (in pixels) for all characters to be drawn. This number, when multiplied by `#lines`, determines the total height (in pixels) of the active text area.

This word *must* be set to the appropriate value if you wish to use *any* `fb1-` or `fb8-` utility routines or `>font`. This can be done with `to`, but is normally done by calling `set-font`.

chars

stack: (nu1 -- nu2)
code: 66

nu2 is the result of multiplying *nu1* by `/c`, the length in bytes of a byte. This is useful for converting an index into a byte offset.

char-width

stack: (-- width)
code: 16D

A `value`, containing the standard width (in pixels) for all characters to be drawn. This number, when multiplied by `#columns`, determines the total width (in pixels) of the active text area.

This word *`must`* be set to an appropriate value if you want to use *any* `fb1-` or `fb8-` utility routines. This can be done with `to`, but is normally done by calling `set-font`.

child

stack: (phandle.parent -- phandle.child)
code: 23B

Returns the phandle of the package that is the first child of the package *phandle.parent*.

`child` returns zero if the package *phandle.parent* has no children.

You will generally use `child`, together with `peer`, to enumerate (possibly recursively) the children of a particular device. One common use could be for bus adapter device drivers to use the phrase `my-self ihandle>phandle` to develop the *phandle.parent* argument.

For example:

```
: my-children ( -- )  \ shows phandles of all children
   my-self ihandle>phandle child  ( first-child )
   begin  ?dup  while  dup .h peer repeat
;
```

"/chosen"

The standard node containing properties describing parameters chosen or specified at run-time for this system. The value of the `name` property of this node is "`chosen`". The remaining properties of this node are:

- stdin
- stdout
- bootpath
- bootargs
- memory
- mmu

claim

stack: ([addr …] len … align -- baseaddr …)
code: none

Allocates *len* … (whose format depends upon the package) bytes of addressable resource. If *align* is zero, the allocated range begins at the specified address *addr*. Otherwise, *addr* … (whose format depends upon the package) is not specified and an aligned address is automatically chosen. The alignment boundary is the smallest power of 2 that is greater than or equal to the value of *align*. *baseaddr* … (whose format is the same as *addr*) is the allocated virtual address, and is equal to *addr* if *align* was zero.

Allocates addressable resources with fine-grained control. In general, `claim` is used only to implement system-specific programs. General purpose memory allocation can be accomplished in a portable fashion by `alloc-mem`.

See also: `alloc-mem`, `"available"`, `free-mem`, `release`

clear

stack: (… --)
code: none

Empties the stack. While `clear` is often useful when interactively debugging, it is almost always inappropriate to use `clear` in a program.

close

stack: (--)
code: none

Closes this previously `open`'d device (e.g. turns off the device, disables PCI memory and/or I/O space accesses, clears the PCI bus master enable bit, unmaps the device and de-allocates any resources that were allocated by `open`). When closing an instance chain, a particular instance's `close` method is executed before its parent instances are closed such that the parents' methods can still be used by `close`.

close-dev

stack: (ihandle --)
code: none

Closes the device identified by *ihandle* as well as all of its parents.

close-package

stack: (ihandle --)
code: 206

Closes the package instance identified by *ihandle* by calling that package's `close` method and then destroying the instance.

For example:

```
: tftp-load-avail? ( -- exist? )
   0 0  " obp-tftp" $open-package  ( ihandle )
   dup ihandle>phandle " load" rot
   find-method  if  drop true  else  false  then
   close-package
;
```

See also: `open-package`, `$open-package`

code

stack: (E: … -- ???)
(C: "new-name< >" -- code-sys)
code: none

Begins the creation of a machine-code command called *new-name*. Interpret the commands which follow as assembler mnemonics until `c;` or `end-code` is encountered.

If the assembler is not installed, code is still present. However, machine code must be hand-assembled and entered into the dictionary by value with `c,`, `w,`, `l`, or `,` .

column# "column number"

stack: (-- column#)
code: 153

A `value`, set and controlled by the terminal emulator, that contains the current horizontal position of the text cursor. A value of 0 represents the leftmost cursor position of the text window, *not* the leftmost pixel of the framebuffer.

`column#` can (and should) be looked at as needed if your FCode Program is implementing its own set of framebuffer primitives.

For example:

```
: set-column  ( column# -- )
   0 max  #columns  1- min  to column#
;
```

See also: `window-left`

#columns "number columns"

stack: (-- columns)
code: 151

This is a `value` that returns the number of columns of text in the text window i.e. the number of characters in a line, to be displayed using the boot ROM's terminal emulator.

`#columns` *must* be set to a proper value in order for the terminal emulator to function correctly. The `open` method of any package that uses the terminal emulator package must set `#columns` to the desired width of the text region. This can be done with `to`, or it can be handled automatically as one of the functions performed by `fb1-install` or `fb8-install`.

See also: `is-install`, `fb1-install`, `fb8-install`, `to`

comp

stack: (addr1 addr2 len -- n)
code: 7A

Compares two strings of length *len* starting at addresses *addr1* and *addr2* and continuing for *len* bytes. *n* is 0 if the arrays are the same. *n* is 1 if the first differing character in the array at *addr1* is numerically greater than the corresponding character in the array at *addr2*. *n* is -1 if the first differing character in the array at *addr1* is numerically less than the corresponding character in the array at *addr2*.

For example:

```
ok " this" drop " that" comp .h
1
ok " thisismy" drop " this" comp .h
0
ok " thin" drop " this" comp .h
ffffffff
```

"compatible"

This standard property specifies a list of devices with which this device is compatible. This property is used by client programs to determine the appropriate driver for the associated device in those cases where the client program does not have a driver matching the value of the `"name"` property.

The property format is identical to the format for the `"name"` property.

For example:

```
" AAPL,dev-name" encode-string
" INTL,my-dev" encode-string encode+
" RST,dev21-type4" encode-string encode+
" compatible" property
```

Please note that you must ensure compatibility with another device's driver prior to using the `"compatible"` property; Open Firmware makes no attempt to cross-check the correctness of the claim of compatibility.

compile

stack: (--)
code: none

Compiles the following command at run time.

Included for compatibility. `Postpone` is preferred for new code.

compile,

stack: (xt --)
code: DD

Compiles the behavior of the word given by *xt*.

[compile]

stack: ([old-name< >] --)
code: none

Compiles the immediately-following command.

Included for compatibility. `postpone` is preferred for new code.

config-b@

stack: (config-addr -- data)
code: none

Performs an 8-bit Configuration Read from the configuration space of a PCI device.

config-addr refers to the *phys.hi* cell of a PCI address (as returned by `my-space`).

This is a method of a PCI bus node. Consequently, the method is typically accessed by PCI child devices with `$call-parent`.

See also: `config-b!`, `config-l@`, `config-l!`, `config-w@`, `config-w!`

config-b!

stack: (data config-addr --)
code: none

Performs an 8-bit Configuration Write from the configuration space of a PCI device.

config-addr refers to the *phys.hi* cell of a PCI address (as returned by `my-space`).

This is a method of a PCI bus node. Consequently, the method is typically accessed by PCI child devices with `$call-parent`.

See also: `config-b@`, `config-l@`, `config-l!`, `config-w@`, `config-w!`

config-l@

stack: (config-addr -- data)
code: none

Performs a 32-bit Configuration Read from the configuration space of a PCI device.

config-addr refers to the *phys.hi* cell of a PCI address (as returned by `my-space`). *config-addr* must be 32-bit aligned.

This is a method of a PCI bus node. Consequently, the method is typically accessed by PCI child devices with `$call-parent`. Of course, the method can also be accessed from the bus node itself. For example, to read the Device ID and Vendor ID for all possible slots on the PCI bus and print them in a formatted listing:

```
ok hex
ok select /pci
ok 20 0 do i 2 u.r i 800 * config-l@ 9 u.r cr loop
ok unselect-dev
```

See also: `config-b@`, `config-b!`, `config-l!`, `config-w@`, `config-w!`

config-l!

stack: (data config-addr --)
code: none

Performs a 32-bit Configuration Write from the configuration space of a PCI device.

config-addr refers to the *phys.hi* cell of a PCI address (as returned by `my-space`). *config-addr* must be 32-bit aligned.

This is a method of a PCI bus node. Consequently, the method is typically accessed by PCI child devices with `$call-parent`.

See also: `config-b@`, `config-b!`, `config-l@`, `config-w@`, `config-w!`

config-w@

stack: (config-addr -- data)
code: none

Performs a 16-bit Configuration Read from the configuration space of a PCI device.

config-addr refers to the *phys.hi* cell of a PCI address (as returned by `my-space`). *config-addr* must be 16-bit aligned.

This is a method of a PCI bus node. Consequently, the method is typically accessed by PCI child devices with `$call-parent`. For example, to read the Device ID:

```
my-space 2+ " config-w@" $call-parent ( device-id )
```

See also: `config-b@`, `config-b!`, `config-l@`, `config-l!`, `config-w!`

config-w!

stack: (data config-addr --)
code: none

Performs a 16-bit Configuration Write from the configuration space of a PCI device.

config-addr refers to the *phys.hi* cell of a PCI address (as returned by `my-space`). *config-addr* must be 16-bit aligned.

This is a method of a PCI bus node. Consequently, the method is typically accessed by PCI child devices with `$call-parent`. For example, to enable I/O Space accesses:

```
my-space 4 + dup " config-w@" $call-parent ( config-addr value )
1 or swap " config-w!" $call-parent
```

See also: `config-b@`, `config-b!`, `config-l@`, `config-l!`, `config-w@`

constant

stack: (x "new-name< >" --) (E: -- value)
code: none
generates: `new-token|named-token|external-token b(constant)`

Creates a named constant. The name is initially created with:

```
456 constant purple
```

where `456` is the desired value for `purple`.

Later occurrences of `purple` will leave the correct value on the stack. `constant` values should never be changed by the program. If you wish to change the value of a `constant` in a program, you should use `value` instead of `constant`.

2constant

stack: (x1 x2 "new-name< >" --) (E: -- x1 x2)
code: none

Creates a named two-number constant.

control

stack: ([text< >] -- char)
code: none
generates: `b(lit) 00 00 00 xx-byte`

Causes the compiler/interpreter to interpret the next letter as a control-code. For example:

```
control c  ( equals 03 )
```

count

stack: (pstr -- str len)
code: 84

Converts a packed string into a byte-array format. *pstr* is the address of a packed string, where the byte at address *pstr* is the length of the string and the string itself starts at address *pstr*+1.

Packed strings are generally not used in FCode. Virtually all string operations are in the "str len" format.

For example:

```
h# 100 alloc-mem constant my-buff
" This is a string" my-buff pack ( pstr ) count type
```

cpeek

stack: (addr -- false | byte true)
code: 220

Tries to read the 8-bit byte at address *addr*. Returns the data and *true* if the access was successful. A *false* return indicates that a read access error occurred.

See also: rb@

cpoke

stack: (byte addr -- okay?)
code: 223

Attempts to write the 8-bit byte at address *addr*. Returns *true* if the access was successful. A *false* return indicates that a write access error occurred.

Note – cpoke may be unreliable on bus adapters that buffer write accesses.

See also: rb!

cr

stack: (--)
code: 92

A defer word used to terminate the line on the display and go to the next line. The default implementation transmits a carriage return and line feed to the display, clears #out and adds 1 to #line.

Use cr whenever you want to start a new line of output, or to force the display of any previously buffered output text. This forcing is valuable for outputting error messages, to ensure that the error message is sent *before* any system crash.

For example:

```
: show-info ( -- )
   ." This is the first line of output " cr
   ." This is the second line of output " cr
;
```

(cr

stack: (--)
code: 91

Outputs only the carriage return character (`carret`, 0x`0D`). The most common use of `(cr` is for reporting the progress of a test that has many steps. By using `(cr` instead of `cr`, the progress report appears on a single line instead of scrolling.

create

stack: ("new-name< >" --) (E: -- addr)
code: none
generates: `new-token|named-token|external-token` `b(create)`

Creates the name *new-name*. When *new-name* is subsequently executed, it returns the address of memory immediately following *new-name* in the dictionary. You can use `create` to build an array-type structure, as:

```
create green 77 c, 23 c, ff c, ff c, 47 c, 22 c, …
```

Later execution of `green` leaves the address of the first byte of the array (here, the address of the byte “77”) on the stack. The returned address will be two-byte aligned.

`create` may *not* be used within definitions in an FCode Program. The common Forth construct `create…does>` is not supported.

See also: `$create`

$create “dollar create”

stack: (name-str name-len --) (E: -- addr)
code: none

Like `create` but takes a name string from the stack.

ctrace “see trace”

stack: (--)
code: none

Displays the subroutine call stack that was in effect when the program state was saved (i.e. when the program was suspended). The format of the display is implementation dependent.

d# “dee number”

stack: ([number< >] -- n)
code: none
generates: `b(lit)` `xx-byte` `xx-byte` `xx-byte` `xx-byte`

Causes the compiler/interpreter to interpret the next number in decimal (base 10), regardless of any previous settings of `hex`, `decimal` or `octal`. Only the immediately following number is affected, the default numeric base setting is unchanged. For example:

```
hex
d# 100  ( equals decimal 100 )
100     ( equals decimal 256 )
```

See also: `h#`, `o#`

d+

“dee plus”

stack: (d1 d2 -- d.sum)
code: D8

Adds two 64-bit numbers, leaving the 64-bit sum on the stack.

For example:

```
ok 1234.0000 0056.7800 9abc 3400.009a d+ .s
1234.9abc 3456.789a
```

See also: `um*`, `um/mod`

d-

“dee minus”

stack: (d1 d2 -- d.diff)
code: D9

Subtracts two 64-bit numbers, leaving the 64-bit result on the stack.

For example:

```
ok 0 6 1 0 d- .s
ffff.ffff 5
ok 4444.8888 aaaa.bbbb 2222.1111 5555.2222 d- .s
2222.7777 5555.9999
```

See also: `um*`, `um/mod`

.d

“dot dee”

stack: (n --)
code: none
generates: `base @ swap d# 10 base ! . base !`

Displays *n* in decimal with a trailing space. The value of `base` is not permanently affected.

See also: `.h`

"deblocker"

This is the standard package to assist in the implementation of byte-oriented `read` and `write` methods for block-oriented or record-oriented storage devices such as disks and tapes. The package provides a layer of buffering to implement a high-level byte-oriented interface above a low-level block-oriented interface.

The `deblocker` package implements the following methods:

- `open ( -- okay? )`

  Prepare this device for subsequent use.
- `close ( -- )`

  Close this previously-`open`'d device.
- `read ( addr len -- actual )`

  Read from device into the specified memory buffer, returning the number of bytes actually read.
- `write ( addr len -- actual )`

  Write to the device from the specified memory buffer, returning the number or bytes actually written.
- `seek ( offset file# -- status )`

  Set the position at which the next `read` or `write` will take place, returning -1 if the operation fails and either 0 or 1 if the operation succeeds.

Any package that uses the `deblocker` package must define the following interface methods:

- `block-size ( -- block-len )`

  Return the "granularity" of the device in bytes.
- `dma-alloc ( … size -- virt )`

  Allocate *size* bytes of contiguous memory with the DMA address space of the device bus, returning the virtual address *virt*.
- `max-transfer ( -- max-len )`

  Return the size of the largest possible transfer in bytes rounded down to an integer multiple of `block-size`.
- `read-blocks ( addr block# #blocks -- #read )`

  Read *#blocks*, starting from *block#*, from the device into the memory beginning at *addr*, returning the number of blocks actually read.
- `write-blocks ( addr block# #blocks -- #written )`

  Write *#blocks*, starting from *block#*, from memory beginning at *addr* to the device, returning the number of blocks actually written.

debug

stack: ("old-name< >" --)
code: none

Marks the command `old-name` for debugging. Subsequent attempts to execute `old-name` cause entry into the Forth source-level debugger. An Open Firmware system that implements the source-level debugger must allow at least one command to be marked at any given moment and may allow several commands to be marked for debugging simultaneously.

During the execution of a debugged command, the name of the command that is about to be executed and the contents of the stack are displayed before the execution of each command called by the debugged command.

Debugging occurs in either "step mode" or "trace mode" as controlled by the stepping commands, `stepping` and `tracing`.

In “step mode” each “step” represents the execution of a single Forth word. The user controls the progress of execution with the following keystrokes:

Table 35 “Step” Mode Commands for the Source-Level Debugger

Keystroke	Description
`<space-bar>`	Execute the word just displayed and proceed to the next word.
`d`	“Down a level”. Mark for debugging the word whose name was just displayed, then execute it.
`u`	“Up a level”. Un-mark the word being debugged, mark its caller for debugging, and finish executing the word that was previously being debugged.
`c`	“Continue”. Switch from stepping to tracing, thus tracing the remainder of the execution of the word being debugged.
`f`	Start a subordinate Forth interpreter with which Forth commands can be executed normally. When that interpreter is terminated (with `resume`), control returns to the debugger at the place where the `f` command was executed.
`resume`	Exit from a subordinate interpreter, and go back to the stepper (See the preceding `f` command.)
`q`	“Quit”. Abort the execution of the word being debugged and all its callers and return to the command interpreter.

Table 36 FirmWorks/Sun “Step” Mode Extensions

Keystroke	Description
`g`	“Go.” Turn off the debugger and continue execution.
`h`	“Help”. Display symbolic debugger documentation.
`?`	“Short Help”. Display brief symbolic debugger documentation.
`s`	“see”. Decompile the word being debugged.
`$`	Non-destructively display the *address,len* on top of the stack as a text string.
`(`	Moves the beginning of the debug region to the current position in the word being debugged.
`)`	Moves the end of the debug region to the current position in the word being debugged.
`*`	Expands the debug region to include the entire word.
`<`	Moves the beginning of the debug region to just after the current position in the word being debugged, and moves the end of the debug region to the end of that word. This is useful for skipping past the end of a loop — step to the word that ends the loop then type `<` .

In “trace mode”, execution continues automatically with each of the words that are called by the marked word. As the words are executed, calling information is displayed.

Debug mode can be turned off with the `debug-off` command.

The system does not necessarily operate at full speed when one or more commands are marked for debugging.

Debugging basic Forth commands (which could have been used in the implementation of `debug`) is not recommended. The system may ignore requests to debug words that are “unsafe” to debug.

See also: `(debug`, `debugging`, `debug(`, `)debug`, `debug-me`, `debug-off`, `stepping`, `tracing`, `resume`

(debug

stack: (xt --)
code: none

Marks the command indicated by *xt* for debugging.

debug(

stack: (--)
code: none

A debugger extension provided by some implementations (e.g. FirmWorks/Sun).

`debug(` can be compiled into a word. When `debug(` is executed, it invokes the debugger with the debugger’s scope beginning just after the call to `debug(` and ending at the end of the word.

)debug

stack: (--)
code: none

A debugger extension provided by some implementations (e.g. FirmWorks/Sun).

`)debug` can be compiled into a word. When `)debug` is executed, it moves the end of the debugger’s scope to just after the call to `)debug`.

debug-me

stack: (--)
code: none

A debugger extension provided by some implementations (e.g. FirmWorks/Sun).

`debug-me` can be compiled into a word. When `debug-me` is executed, it invokes the debugger on the word containing it, and makes the debugger’s scope the entire word containing `debug-me` even though debugging is not triggered until `debug-me` is first executed.

debug-off

stack: (--)
code: none

Turns off the Forth source-level debugger.

debugging

stack: ("old-name< >" --)
code: none

A debugger extension provided by some implementations (e.g. FirmWorks/Sun).

A shorthand way to mark a word for debugging and execute it. `debugging` *foo* is equivalent to `debug` *foo foo*.

decimal

stack: (--)
code: none

If used outside of a definition, commands the tokenizer program to interpret subsequent numbers in decimal (base 10).

If used within a definition, appends the phrase `10 base !` to the FCode Program that is being created thus affecting later numeric output when the FCode Program is executed.

See also: `base`

decode-bytes

stack: (prop-addr1 prop-len1 data-len -- prop-addr2 prop-len2 data-addr data-len)
code: none
generates: `>r over r@ + swap r@ - rot r>`

Decodes *data-len* bytes from a property value array and returns the remainder of the array and the decoded byte array.

decode-int

stack: (prop-addr1 prop-len1 -- prop-addr2 prop-len2 n)
code: 21B

Decodes a number from the beginning of a property value array and returns the remainder of the property value array and the number *n*.

For example:

```
: show-clock-frequency ( -- )
   " clock-frequency" get-inherited-property  0=  if
      ." Clock frequency: " decode-int .h cr 2drop
   then
;
```

decode-phys

stack: (prop-addr1 prop-len1 -- prop-addr2 prop-len2 phys.lo … phys.hi)
code: 128

Decodes a unit address from a property value array and returns the remainder of the array and the decoded list of address components. The number of cells in the list *phys.lo … phys.hi* is determined by the value of the "`#address-cells`" property of the parent node.

decode-string

stack: (prop-addr1 prop-len1 -- prop-addr2 prop-len2 str len)
code: 21C

Decodes a string from the beginning of a property value array and returns the remainder of the property value array and the string *str len*.

For example:

```
: show-model ( -- )
   " model" get-my-property  0=  if  decode-string type 2drop  then
;
```

decode-unit

stack: (addr len -- phys.lo … phys.hi)
code: none

A static method that converts a text representation of a unit-address into a numerical representation of a physical address within the address space defined by this device node. The number of cells in the list `phys.lo` … `phys.hi` is determined by the value of the `"#address-cells"` property of the parent node.

default-font

stack: (-- addr width height advance minchar #glyphs)
code: 16A

Returns all the necessary information about the character font that is built into the boot ROM. This font defines the appearance of every character to be displayed. To load this font, simply pass these parameters to `set-font`, with:

```
default-font set-font
```

The actual parameters returned by `default-font` are:

addr - The address of the beginning of the built-in font table

width - The width of each character in pixels

height - The height of each character in pixels

advance- The separation (in bytes) between each scan line entry

minchar- The ASCII value for the first character actually stored in the font table.

#glyphs - The total number of characters stored in the font table.

defer

stack: ("new-name< >" --) (E: -- ???)
code: none
generates: `new-token|named-token|external-token b(defer)`

Creates a command *new-name* that is a `defer` word i.e. a word whose behavior can be altered with `to`. *new-name* is initially created with execution behavior that indicates that it is an uninitialized `defer` word. For example:

```
ok defer blob
ok blob
<--deferred word not initialized
```

Later, this behavior can then be altered to be that of another existing word by placing that second word’s execution token on the stack and loading it into *new-name* with `to`.

For example:

```
['] foobar to blob
```

`defer` words are useful for generating recursive routines. For example:

```
defer hold2  \ Will execute action2
: action1
  …
  hold2 ( really action2 )
  …  ;
: action2
  …
  action1
  … ;
' action2 to hold2
```

`defer` words can also be used for creating words with different behaviors depending on your needs. For example:

```
defer .special ( n -- ) \ Print a value, using special techniques
: print-em-all ( -- )
  … .special
  … .special
  … .special
;

( .d prints in decimal
( .h prints in hexadecimal )
( .sp prints in a custom format )
: print-all-styles
  ['] .d to  .special print-em-all
  ['] .h to  .special print-em-all
  ['] .sp to .special print-em-all
;
```

In FCode source, `defer` cannot appear inside a colon definition.

See also: `behavior`, `to`

delete-characters

stack: (n --)
code: 15E

Deletes *n* characters to the right of the cursor.

`delete-characters` is one of the `defer` words of the display device interface. The terminal emulator package executes `delete-characters` when it has processed a character sequence that requires the deletion of characters to the right of the cursor. The cursor position is unchanged, the cursor character and the first *n*-1 characters to the right of the cursor are deleted. All remaining characters to the right of the cursor, including the highlighted character, are moved left by *n* places. The end of the line is filled with blanks.

This word is initially empty, but *must* be loaded with an appropriate routine in order for the terminal emulator to function correctly. This can be done with `to`, or it can be loaded automatically with `fb1-install` or `fb8-install` (which loads `fb1-delete-characters` or `fb8-delete-characters`, respectively).

See also: `fb8-install`, `to`

delete-lines

stack: (n --)
code: 160

Deletes *n* lines at and below the cursor line.

`delete-lines` is one of the `defer` words of the display device interface. The terminal emulator package executes `delete-lines` when it has processed a character sequence that requires the deletion of lines of text below the line containing the cursor. All lines below the deleted lines are scrolled upwards by *n* lines, and *n* blank lines are placed at the bottom of the active text area.

Use this word for scrolling, by temporarily moving the cursor to the top of the screen and then calling `delete-lines`.

This word is initially empty, but *must* be loaded with an appropriate routine in order for the terminal emulator to function correctly. This can be done with `to`, or it can be loaded automatically with `fb1-install` or `fb8-install` (which load `fb1-delete-lines` or `fb8-delete-lines`, respectively).

See also: `fb8-install`, `to`

delete-property

stack: (name-str name-len --)
code: 21E

Deletes the property named by *name-str name-len* in the active package, if such a property exists.

For example:

```
: unmap-me ( -- )
   my-reg my-size " map-out"  $call-parent
   " address" delete-property
;
```

depth

stack: (-- u)
code: 51

u is the number of entries contained in the data stack, not counting itself. Note that when an FCode Program is called, there could be other items on the stack from the calling program.

`depth` is especially useful for before/after stack depth checking, to determine if the stack was corrupted by a particular operation.

"depth"

This standard property is associated with `display` devices. The property value is an integer (encoded with `encode-int`) that specifies the number of bits used to describe each pixel of the display.

See also: `property`

dev

stack: ("device-specifier<eol>" --)
code: none

Makes the specified device node the active package by parsing *device-specifier* delimited by the end of line. Perform the equivalent of `find-device` with *device-specifier* as its argument.

For example:

```
ok dev device-specifier <eol>
```

See also: `device-end`

devalias

stack: ("{alias-name}< >{device-path}<eol>" --)
code: none

Creates a device alias, or displays the current alias(es).

If *alias-name* and *device-path* are specified, `devalias` creates a device alias named *alias-name* representing *device-path*. If an alias with the same name already exists, the new value supersedes the old.

If only *alias-name* is specified, `devalias` displays the device path corresponding to *alias-name* (if this alias exists).

If nothing is specified after devalias, `devalias` displays all currently existing device aliases.

device-end

stack: (--)
code: none

Unselects the active package leaving none selected.

device-name

stack: (str len --)
code: 201

Creates a `"name"` property with the given string value. For example:

```
" AAPL,zebra" device-name
```

This is equivalent to:

```
" AAPL,zebra" encode-string " name" property
```

See also: `"name"`, `property`

device-type

stack: (str len --)
code: 11A

This is a shorthand word for creating a `"device_type"` property. This property is essential for any plug-in PCI device that will be used during booting, as it tells the boot ROM which type of boot device it is. An example usage is:

```
" display" device-type
```

This is exactly equivalent to the following:

```
" display" encode-string " device_type" property
```

Note the spelling difference between the FCode command `device-type` (hyphen) and the `"device_type"` property (underscore).

See also: `"device_type"`, `property`

"device_type"

This property declares the type of this plug-in device. The type need not be declared, unless this device is intended to be usable for booting. If this property is declared, using one of the following key values, the FCode program *must* follow the required conventions for the specified device type. Used as:

```
" display" encode-string " device_type" property
```

Defined values for this property are `"block"`, `"byte"`, `"display"`, `"memory"`, `"network"`, `"pci"` and `"serial"`.

See also: `device-type`, `property`, Chapter 5 “Properties

diag-device

stack: (-- dev-str dev-len)
code: none

The value of this configuration variable is a string describing the default device-name and any required filename to be used by `boot` if `diagnostic-mode?` is `true`. *dev-string* is a device-specifier or a list of device-specifiers, as described in `load`.

The suggested default value is “net”.

diag-file

stack: (-- arg-str arg-len)
code: none

The value of this configuration variable is a string describing the default arguments to be used by `boot` if `diagnostic-mode?` is `true`.

The suggested default value is “diag”.

diagnostic-mode?

stack: (-- diag?)
code: 120

`diagnostic-mode?` controls several aspects of machine function.

During booting, `diagnostic-mode?` controls the choice of boot device and boot file, if not specified in the boot command. If `diagnostic-mode?` is true, the default boot device is specified by `diag-device` and the default boot file is specified by `diag-file`. If `diagnostic-mode?` is false, the default boot device is specified by `boot-device` and the default boot file is specified by `boot-file`.

During machine power-on, `diagnostic-mode?` controls the extent of system selftest and controls the amount of informative messages displayed. If `diagnostic-mode?` is true, more extensive tests are performed and more messages are displayed. The details of selftest, however, are implementation-dependent.

FCode Programs can use `diagnostic-mode?` to control the extent of the selftests performed. While the specifics of use are controlled by the FCode Program itself, the recommended use is described in the preceding paragraph. In other words, if `diagnostic-mode?` is true, more extensive tests are performed and more messages are displayed.

For example:

```
diagnostic-mode?
if    do-extended-tests
else  do-normal-tests
then
```

FCode should not generate character output during probing unless `diagnostic-mode?` is `true`, or unless an error is encountered. Error output during probing typically goes to the system serial port.

`diagnostic-mode?` will return true if any of the following conditions are met:

- `diag-switch?` is true.
- A machine diagnostic switch (system-dependent) is ON.
- Other system-dependent indicators request extensive diagnostics.

See also: `diag-switch?`

diag-switch?

stack: (-- diag?)
code: none

This configuration variable is a boolean describing whether more extensive diagnostics should be run. If `diag-switch?` is true, `diagnostic-mode?` returns true.

The suggested default value of `diag-switch?` is “false”.

Note that `diag-switch?` true implies `diagnostic-mode?` true, but `diag-switch?` false does not imply `diagnostic-mode?` false. Other system-dependent mechanisms can cause `diagnostic-mode?` to be true.

See also: `diagnostic-mode?`

digit

stack: (char base -- digit true | char false)
code: A3

If the character *char* is a digit in the specified base, returns the numeric value of that digit under *true*, else returns the character under *false*. Appropriate characters are hex `30-39` (for digits 0-9) and hex `61-66` (for digits `a-f`), depending on `base`.

For example:

```
: probe-slot ( slot# -- ) … ;
: probe-slots  ( addr cnt -- )
   bounds  ?do
      i c@  d# 16  digit  if  probe-slot  else  drop  then
loop
;
```

dis

stack: (addr --)
code: none

Begins disassembling at the given address. The format of the disassembly, and the conditions for stopping disassembly, are system-dependent.

See also: `+dis`

+dis

stack: (--)
code: none

Continues disassembling where `dis` or `+dis` last stopped.

See also: `dis`

"disk-label"

This is the standard package that interprets the disk label including any "partitioning" information. This package is used by `block` device drivers.

The `disk-label` package uses the `read` and `seek` methods of the package that `open`'d it. `disk-label` implements the following methods:

- `open  ( -- okay? )`

  Prepare this device for subsequent use.

- `close  ( -- )`

  Close this previously-`open`'d device.

- `load  ( addr -- size )`

  Load a client program from device to memory.

- `offset  ( d-rel -- d.abs )`

  Convert a partition-relative disk position to an absolute position.

"display"

This is the standard property value of the `"device_type"` property for user output devices with randomly-addressable pixels (i.e.graphic output display device devices). `"display"` devices can be used for console output.

Devices of type `"display"` must implement the following methods:

- `open`
- `close`
- `write`

  Display the sequence of *len* characters beginning at *addr*, interpreting command sequences as defined by Annex B of *IEEE Standard 1275-1994*.

- `draw-logo`

If an unexpected system reset can cause the display to become invisible (e.g. the video is turned off) and the display can be restored to visibility without performing memory mapping or memory allocation operations, the `restore` method should also be implemented.

`display` devices may also implement additional device-specific methods.

`display` packages typically use the terminal emulator support package to process ANSI X3.64 escape sequences for the write method. “Dumb” frame-buffer devices typically use either the "fb1" or the "fb8" support package to implement the “Character Map” defer words interface. More complicated display devices, such as those with hardware acceleration, typically implement that interface directly.

display-status

stack: (n --)
code: 121

Displays the results of some test. The method of display is system-dependent. This FCode is obsolete and should not be used.

dl “dee el”

stack: (--)
code: none

Downloads and execute Forth text, end with Control-D.

Receives text from the current input source and stores it in a buffer, until an EOT (0x04, or control-D) character is received. Does not store the EOT character.

After reception, evaluates the contents of the buffer as with the `eval` command. The buffer size is system-dependent and is at least 4096 characters.

`dl` is typically used with a serial line as the current input source. After issuing the `dl` command, the user typically issues commands to another computer to cause the desired Forth text (such as a text file) to be sent over the serial line, followed by the EOT (0x04, or control-D) character.

`fload` commands that are embedded in the downloaded text will not be processed correctly. See “Downloading Multiple Files with dl and fload” on page 28 for more information.

See also: `fload`

dma-alloc

stack: (… size -- virt)
code: none

Allocates *size* bytes of memory, contiguous within the direct-memory-access address space of the device bus, suitable for direct memory access by a “bus master” device. The memory is allocated according to the most stringent alignment requirements for the bus. Returns the virtual address *virt*. That virtual address is suitable for CPU access to the allocated region, but, in general, `dma-map-in` must be used to convert it to an address suitable for direct memory access by the bus-master device.

The ellipsis in the stack diagram indicates that some memory-mapped buses may require additional mapping space parameters. See the Open Firmware binding for the bus in question. (In the case of the PCI bus, no additional parameters required.)

If the requested operation cannot be performed, a `throw` is called with an appropriate error message, as with `abort"`. Consequently, out-of-memory conditions can be detected and handled properly in the code with the phrase `['] dma-alloc catch`.

Drivers will normally use the `dma-alloc` method of their parent:

```
" dma-alloc" $call-parent
" dma-map-in" $call-parent
```

For example:

```
: my-dma-alloc  ( -- )
   my-size " dma-alloc"  $call-parent   ( vaddr )
   to my-vaddr
;
```

See also: `dma-map-in`, `dma-free`, Appendix C, “PCI Bus Binding to Open Firmware”

dma-free

stack: (virt size --)
code: none

Frees *size* bytes of memory at virtual address *virt* that were previously allocated with `dma-alloc`.

dma-map-in

stack: (… virt size cacheable? -- devaddr)
code: none

Converts the virtual address range *virt size* that was previously allocated by the `dma-alloc` method into an address suitable for DMA on the device bus. Returns this address *devaddr*.

`dma-map-in` can also be used to map application-supplied data buffers for DMA use on the bus, if possible.

The flag *cacheable?* should be nonzero if you would like to make use of caches for the DMA buffer if they are available.

Immediately after `dma-map-in` has been executed, the contents of the address range as seen by the processor (the processor’s “view”) is the same as the contents as seen by the device that performs the DMA (the device's "view"). After the DMA device has performed DMA or the processor has performed a write to the range in question, the contents of the address range as seen by the processor (the processor's "view") is not necessarily the same as the contents as seen by the device that performs the DMA (the device's "view"). The two views can be made consistent by executing `dma-map-out` or `dma-sync`.

The ellipsis in the stack diagram indicates that some memory-mapped buses may require additional mapping space parameters. See the Open Firmware binding for the bus in question. (In the case of the PCI bus, no additional parameters required.)

If the requested operation cannot be performed, a `throw` is called with an appropriate error message, as with `abort"`. Consequently, out-of-memory conditions can be detected and handled properly with the phrase `['] dma-map-in catch`.

See also: `dma-map-out`, Appendix C, "PCI Bus Binding to Open Firmware"

dma-map-out

stack:	(virt devaddr size --)
code:	none

Frees the DMA mapping specified by *virt devaddr size* that was previously created with `dma-map-in`. In addition, flushes all caches associated with that mapping (with `dma-sync`).

dma-sync

stack:	(virt devaddr size --)
code:	none

Flushes any memory caches associated with the DMA mapping *virt devaddr size.*

The parameters *virt devaddr* and *size* need not be identical to the values previously used with `dma-map-in` to obtain the memory cache.

`dma-map-in` and `dma-map-out` must map and unmap memory in identically-sized pieces. However, `dma-sync` can work on smaller pieces within a given mapping. The system will automatically round up the `dma-sync` parameters to the appropriate synchronization boundary (typically a cache line boundary) which is not necessarily the same as the mapping boundary (typically a page boundary).

do

stack:	(C: -- dodest-sys)
	(limit start --) (R: -- sys)
code:	none
generates:	`b(do) +offset`

Begins a counted loop in the form `do…loop` or `do…+loop`. The loop index begins at *start*, and terminates based on *limit*. See `loop` and `+loop` for details on how the loop is terminated. The loop is always executed at least once. For example:

```
8 3 do  i .  loop    \ would print 3 4 5 6 7
9 3 do  i .  2 +loop \ would print 3 5 7
```

`do` may be used either inside or outside of colon definitions.

?do "question do"

stack: (C: -- dodest-sys)
(limit start --) (R: -- sys)
code: none
generates: `b(?do) +offset`

Begin a counted loop in the form `?do`…`loop` or `?do`…`+loop`. The loop index begins at *start*, and terminates based on *limit*. See `loop` and `+loop` for details on how the loop is terminated. Unlike `do`, if *start* is equal to *limit* the loop is executed zero times. For example:

```
8 1 ?do i . loop     \ would print 1 2 3 4 5 6 7
2 1 ?do i . loop     \ would print 1
1 1 ?do i . loop     \ would print nothing
1 1  do i . loop     \ would print 1 2 3 4 5 6 7 8 9 …
```

`?do` can be used in place of `do` in nearly all circumstances. `?do` may be used either inside or outside of colon definitions.

does>

stack: (E: … -- ???)
code: none

Sets the run-time behavior of a `create`…`does>` construct.

draw-character

stack: (char --)
code: 157

A `defer` word that is called by the boot ROM's terminal emulator in order to display a single character on the screen at the current cursor location.

This word is initially empty, but *must* be loaded with an appropriate routine in order for the terminal emulator to function correctly. This can be done with `to`, or it can be loaded automatically with `fb1-install` or `fb8-install` (which loads `fb1-draw-character` or `fb8-draw-character`, respectively).

draw-logo

stack: (line# addr width height --)
code: 161

A `defer` word that is called by the system to display the power-on logo (the graphic displayed on the left side during power-up, or by `banner`).

This word is initially empty, but *must* be loaded with an appropriate routine in order for the terminal emulator to function correctly. This can be done with `to`, or it can be loaded automatically with `fb1-install` or `fb8-install` (which load `fb1-draw-logo` or `fb8-draw-logo`, respectively).

It is possible to pack a custom logo into the FCode ROM and then re-initialize `draw-logo` to output the custom logo instead.

`draw-logo` is called by the system using the following parameters:

line# - The text line number at which to draw the logo. For general use, see Appendix B, "FCode Memory Allocation".

addr - The address of the logo template to be drawn. In practice, this will always be either the address of the `oem-logo` field in NVRAM, the address of a custom logo in the FCode ROM, or the address of the built-in system logo. In either case, the logo is a bit array of 64x64 (decimal) pixels (512 bytes). The most significant bit (MSB) of the first byte represents the upper-left pixel; (MSB-1) represents the next pixel to the right, and so on. A bit value of 1 means that pixel will be painted.

width - The width of the passed-in logo (in pixels).

height- The height of the passed-in logo (in pixels).

draw-logo

stack: (line# addr width height --)
code: none

A package method that draws a logo on an output device. The arguments and semantics of this method are identical to those of the `draw-logo` FCode Function.

`is-install` automatically creates an implementation of this method that executes the `draw-logo` `defer` word.

See also: `"display"`, `banner`, `draw-logo` (FCode Function)

drop

stack: (x --)
code: 46

Removes one item from the stack.

2drop

stack: (x1 x2 --)
code: 52

Removes two items from the stack.

3drop

stack: (x1 x2 x3 --)
code: none
generates: `drop` `2drop`

Removes three items from the stack.

dump

stack: (addr len --)
code: none

Displays *len* bytes of memory starting at *addr.*

dup “dupe”

stack: (x -- x x)
code: 47

Duplicates the top stack item.

2dup “two dupe”

stack: (x1 x2 -- x1 x2 x1 x2)
code: 53

Duplicates the top two stack items.

3dup “three dupe”

stack: (x1 x2 x3 -- x1 x2 x3 x1 x2 x3)
code: none
generates: `2 pick 2 pick 2 pick`

Duplicates the top three stack items.

?dup “question dupe”

stack: (x -- 0 | x x)
code: 50

Duplicates the top stack item unless it is zero.

else

stack: (C: orig-sys1 -- orig-sys2)
(--)
code: none
generates: `bbranch +offset b(>resolve)`

Begin the `else` clause of an `if`...`else`...`then` statement. See `if` for more details.

emit

stack: (char --)
code: 8f

A `defer` word that outputs the indicated ASCII character. For example, `h# 41 emit` outputs an “A”, `h# 62 emit` outputs a “b”, `h# 34 emit` outputs a “4”.

emit-byte

stack: (FCode# --)
code: none
generates: `n`

An FCode-only command used to manually output a desired byte of FCode. Use it together with `tokenizer[` as follows:

```
tokenizer[
  44 emit-byte 20 emit-byte
]tokenizer
```

`emit-byte` would be useful, for example, if you wished to generate a new FCode command that the tokenizer did not understand. This command should be used with caution or else an invalid FCode Program will result.

See also: `tokenizer[`, `]tokenizer`

encode+

stack: (prop-addr1 prop-len1 prop-addr2 prop-len2 -- prop-addr3 prop-len3)
code: 112

Merge two property value arrays into a single property value array. The two input arrays must have been created sequentially with no intervening dictionary allocation or other property value arrays having been created. This can be called repeatedly, to create complex, multi-valued property value arrays for passing to `property`.

For example, suppose you wished to create a property named `myprop` with the following information packed sequentially:

```
"size" 2000 "vals" 3 128 40 22
```

This could be written in FCode as follows:

```
: encode-string,num ( addr len number -- )
  >r encode-string
  r> encode-int encode+
;
" size" 2000 encode-string,num
" vals"    3 encode-string,num encode+
128      encode-int     encode+
40       encode-int     encode+
22       encode-int     encode+
" myprop" property
```

encode-bytes

stack: (data-addr data-len -- prop-addr prop-len)
code: 115

Encodes a byte array into a property value array. The external representation of a byte array is the sequence of bytes itself, with no appended null byte.

For example:

```
my-idprom h# 20 encode-bytes " idprom" property
```

encode-int

stack: (quad -- prop-addr 4)
code: 111

Convert a 4-byte integer into a 4 byte property value array with the most significant byte at the smallest address. Such an array is suitable for passing as a value to `property`. For example:

```
1152 encode-int  " hres" property
```

encode-phys

stack: (phys.lo … phys.hi -- prop-addr prop-len)
code: 113

Encodes a unit-address into a property value array by property encoding the list of cells denoting a unit address in the order of *phys.hi* followed by the encoding of the component that appears on the stack below *phys.hi*, and so on, ending with the encoding of the *phys.lo* component.

The number of cells in the list *phys.lo* … *phys.hi* is determined by the value of the `"#address-cells"` property of the parent node.

For example:

```
my-address my-space encode-phys " resetloc" property
```

encode-string

stack: (str len -- prop-addr prop-len)
code: 114

Converts an ordinary string, such as created by `"`, into a property value array suitable for `property`. For example:

```
" JBB,WMB,GRH" encode-string " authors" property
```

encode-unit

stack: (phys.lo … phys.hi -- unit-str unit-len)
code: none

Converts *phys.lo* … *phys.hi*, the numerical representation, to *unit-string*, the text string representation of a physical address within the address space defined by this device node. The number of cells in the list *phys.lo* … *phys.hi* is determined by the value of the `"#address-cells"` property of this node.

`encode-unit` is a static method.

end0

stack: (--)
code: 00

A word that marks the end of an FCode Program. This word must be present at the end of your program or erroneous results may occur.

If you want to use `end0` inside a colon definition, for example in a conditional clause, use something like:

```
: exit-if-version2  fcode-revision h# 30000 < if  ['] end0 execute  then
;
```

end0

stack: (--)

A User Interface command to cause the command interpreter to ignore the remainder of the input buffer and all subsequent lines from the same input source.

The optional User Interface semantics of this command duplicate the purpose, but not the detailed behavior, of the FCode semantics. The detailed behavior differs because the User Interface command interpreter processes text, while the FCode Evaluator processes byte-coded FCode Programs.

end1

stack: (--)
code: FF

An alternate word for `end0`, to mark the end of an FCode Program. `end0` is recommended.

`end1` is not intended to appear in source code. It is defined as a guard against mis-programmed ROMs. Unprogrammed regions of ROM usually appear as all ones or all zeroes. Having both 0x00 and 0xFF defined as end codes stops the FCode interpreter if it enters an unprogrammed region of a ROM.

endcase

stack: (C: case-sys --)
(sel --)
code: none
generates: `b(endcase)`

Marks the end of a `case` statement. See `case` for more details.

end-code

stack: (C: code-sys --)
code: none

Ends the creation of a machine-code sequence. No additional assembly language code is assembled.

code-sys is balanced by the corresponding `code` or `label`.

endof

stack: (C: case-sys1 of-sys -- case-sys2)
(--)
code: none
generates: `b(endof) +offset`

Marks the end of an `of` clause within a `case` statement. See `case` for more details.

end-package

stack: (--)
code: none

Closes the device tree entry set up with `begin-package` by performing the following:

- Call `finish-device` to close the child device node.
- Set the working vocabulary to Forth.
- Call `close-dev`.

environment?

stack: (str len -- false | value true)
code: none

Return system information based on input keyword. The exact set of recognized keyword strings is implementation-dependent.

erase

stack: (addr len --)
code: none
generates: `0 fill`

Sets *len* bytes of memory beginning at *addr* to zero. No action is taken if *len* is zero.

erase-screen

stack: (--)
code: 15A

A `defer` word that is called once during the terminal emulator initialization sequence in order to completely clear all pixels on the display. This word is called just *before* `reset-screen`, so that the user doesn't actually see the framebuffer data until it has been properly scrubbed.

This word is initially empty, but *must* be loaded with an appropriate routine in order for the terminal emulator to function correctly. This can be done with `to`, or it can be loaded automatically with `fb1-install` or `fb8-install` (which load `fb1-erase-screen` or `fb8-erase-screen`, respectively).

eval

stack: (… str len -- ???)
code: none
generates: `evaluate`

Synonym for `evaluate`.

evaluate

stack: (… str len -- ???)
code: CD

Takes a string from the stack (specified as an address and a length) and interprets the characters in that string as if they were entered from the keyboard. The overall stack effect depends on the commands being executed. For example:

```
" 4000 20 dump" evaluate
```

`evaluate` can be used to interpret the code contained in a Forth text file that has been loaded into memory . For example:

```
ok 10000 buffer: filebuf
ok " /pci/isa/floppy:,\framus.fth" open-dev ( ihandle )
ok >r filebuf 10000 " read" r@ $call-method ( #read )
ok r> close-dev filebuf swap evaluate
ok
```

You can use `evaluate`, like `$find`, to find and execute Forth commands that are not FCodes.

The same cautions apply to `evaluate` as for `$find` in that programs executing Forth commands are likely to encounter portability problems when moved to other systems.

even

stack:	(n -- n | n+1)
code:	none

Rounds to the nearest even integer >= *n*.

execute

stack:	(… xt -- ???)
code:	1D

Executes the word definition whose execution token is *xt*. An error condition exists if *xt* is not an execution token.

For example:

```
: my-word ( addr len -- )
   ." Given string is: " type cr
;
" great" ['] my-word execute
```

execute-device-method

stack:	(… dev-str dev-len method-str method-len -- … false | ??? true)
code:	none

Executes the named method in the package named *dev-string*. *dev-string* is a device-specifier. Returns *false* if the method could not be executed (i.e. the device-specifier is invalid, or that device has no method with the given name, or execution of that method resulted in an `abort` or `throw`). Otherwise, returns *true* above whatever results were placed on the stack by the execution of the method.

See also: `apply`

"existing"

This property defines the regions of virtual address space managed by the memory management unit (MMU) in whose package this property is defined, without regard to whether or not these regions are currently in use.

The property value is an arbitrary number of (*virtual-address,len*) pairs where:

- *virtual-addr* is one or more integers encoded with `encode-int`.
- *len* is one or more integers, each encoded with `encode-int`.

The encodings of *virtual-addr* and *len* are MMU-specific.

See also: `"available"`, `map`, `modify`, `"reg"`, `translate`, `unmap`

exit

stack: (--) (R: nest-sys --)
code: 33

Compiled within a colon definition. When encountered, execution leaves the current word and returns control to the calling word specified by *nest-sys*. Must be preceded by `unloop` if used within a `do` loop .

For example:

```
: probe-loop  ( addr -- )
   \ Generate a tight probe loop until any key is pressed.
   begin dup l@ drop key?  if  drop exit  then  again
;
: find-value  ( test-value start-addr -- )
   \ Searches up to 100 locations looking for a test value.
   100 bounds do      ( test-value )
      i c@ over = if  ( test-value )
         ." Found at " i . cr  drop unloop exit
      then
   loop               ( test-value )
   . ." not found" cr
;
```

See also: `abort`, `leave`, `unloop`

exit? “exit question”

stack: (-- done?)
code: none

Handles output pagination while providing user control. Returns *true* if the user has requested the cessation of output from the current command.

`exit?` is used inside loops that might send many lines of output to the console. It is typically called once for each line of output.

The precise behavior is implementation-dependent; a typical behavior follows:

- If the value contained in the `#line` variable is greater than a predetermined value (typically returned by a word named `lines/page`) prompt the user with the message:

  `More [<space>,<cr>,q] ?`

  and wait for a character to be typed on the console. If that character is “q” return *true*. If that character is “<cr>” (carriage return) arrange for the next call to `exit?` to prompt the user, and return *false*. If the character is neither “q” or “<cr>” set the contents of `#line` to zero and return *false*.
- If a “q” character has been typed on the console input device since the last time that `exit?` was called return *true*.
- If any other character has been typed, prompt for what to do next, as shown above, and return *false*.
- The typical behavior described above has the following features (assuming that output-generating commands call `exit?` once per line of output):

  a) Output pauses at the end of each page of output, allowing the user to either stop further output (“q”), get one more line output before pausing again (“<cr>”) or continue with the next page of output (“<space>”).

b) The user can stop further output at any time by typing “q”.

c) The user can cause a pause before the end of a page by typing a character other than “q”.

expect

stack: (addr len --)
code: 8A

A `defer` word that receives a line of characters from the keyboard and stores them into memory, performing line editing as the characters are typed. Displays all characters actually received and stored into memory. The number of received characters is stored in `span`.

The transfer begins at *addr* proceeding towards higher addresses one byte per character until either a carriage return is received or until *len* characters have been transferred. No more than *len* characters will be stored. The carriage return is not stored into memory. No characters are received or transferred if *len* is zero.

For example:

```
h# 10 buffer: my-name-buff
: hello ( -- )
   ." Enter Your First name " my-name-buff h# 10 expect
   ." FirmWorks Welcomes " my-name-buff span @ type cr
;
```

We encourage the use of `accept` rather than `expect`.

external

stack: (--)
code: none

After issuing `external`, all subsequent definitions are created so that names are later compiled into RAM, regardless of the value of the NVRAM variable `fcode-debug?`. `external` is used to define the package methods that may be called from software external to the package, and whose names must therefore be present.

`external` stays in effect until `headers` or `headerless` is encountered.

For example:

```
external
: open ( -- ok? ) … ;
```

external-token

stack: (--) (F: /FCode-string FCode#/ --)
code: CA

A token-type that is used to indicate that this word should always be compiled with the name header present. Activated by `external`, all subsequent words are created with `external-token` until deactivated with either `headers` or `headerless`. See `named-token` for more details. This word should never be used in source code.

false

stack: (-- false)
code: none
generates: 0

Leaves the value for `false` (i.e. zero) on the stack.

fb1-blink-screen

stack: (--)
code: 174

The built-in default routine to blink or flash the screen momentarily on a generic 1-bit-per-pixel framebuffer. This routine is loaded into the `defer` word `blink-screen` by calling `fb1-install`.

This routine is invalid unless the FCode Program has called `fb1-install` and has initialized `frame-buffer-adr` to a valid virtual address.

This word is implemented simply by calling `fb1-invert-screen` twice. In practice, this can be quite slow (around one full second). It is quite common for a framebuffer FCode Program to replace `fb1-blink-screen` with a custom routine that simply disables the video for 20 milliseconds or so. For example:

```
: my-blink-screen  ( -- )  video-off  20 ms  video-on  ;
…
fb1-install
…
['] my-blink-screen   to blink-screen
```

fb1-delete-characters

stack: (n --)
code: 177

The built-in default routine to delete *n* characters at and to the right of the cursor, on a generic 1-bit-per-pixel framebuffer. This routine is loaded into the `defer` word `delete-characters` by calling `fb1-install`.

This routine is invalid unless the FCode Program has called `fb1-install` and `set-font` and has initialized `frame-buffer-adr` to a valid virtual address.

The cursor position is unchanged, the cursor character and the next *n*-1 characters to the right of the cursor are deleted, and the remaining characters to the right are moved left by *n* places. The end of the line is filled with blanks.

fb1-delete-lines

stack: (n --)
code: 179

The built-in default routine to delete *n* lines, starting with the line below the cursor line, on a generic 1-bit-per-pixel framebuffer. This routine is loaded into the `defer` word `delete-lines` by calling `fb1-install`.

This routine is invalid unless the FCode Program has called `fb1-install` and `set-font` and has initialized `frame-buffer-adr` to a valid virtual address.

The `n` lines at and below the cursor line are deleted. All lines above the cursor line are unchanged. The cursor position is unchanged. All lines below the deleted lines are scrolled upwards by `n` lines, and `n` blank lines are placed at the bottom of the active text area.

fb1-draw-character

stack: (char --)
code: 170

The built-in default routine for drawing a character on a generic 1-bit-per-pixel framebuffer, at the current cursor location. This routine is loaded into the `defer` word `draw-character` by calling `fb1-install`.

This routine is invalid unless the FCode Program has called `fb1-install` and `set-font` and has initialized `frame-buffer-adr` to a valid virtual address.

If `inverse?` is `true`, then characters are drawn inverted (white-on-black). Otherwise (the normal case) they are drawn black-on-white.

fb1-draw-logo

stack: (line# addr width height --)
code: 17A

The built-in default routine to draw the logo on a generic 1-bit-per-pixel framebuffer. This routine is loaded into the `defer` word `draw-logo` by calling `fb1-install`.

This routine is invalid unless the FCode Program has called `fb1-install` and `set-font` and has initialized `frame-buffer-adr` to a valid virtual address.

See `draw-logo` for more information on the parameters passed.

fb1-erase-screen

stack: (--)
code: 173

The built-in default routine to clear (erase) every pixel in a generic 1-bit-per-pixel framebuffer. This routine is loaded into the `defer` word `erase-screen` by calling `fb1-install`.

This routine is invalid unless the FCode Program has called `fb1-install` and has initialized `frame-buffer-adr` to a valid virtual address.

All pixels are erased (not just the ones in the active text area). If `inverse-screen?` is `true`, then all pixels are set to 1, resulting in a black screen. Otherwise (the normal case) all pixels are set to 0, resulting in a white screen.

fb1-insert-characters

stack: (n --)
code: 176

The built-in default routine to insert *n* blank characters to the right of the cursor, on a generic 1-bit-per-pixel framebuffer. This routine is loaded into the `defer` word `insert-characters` by calling `fb1-install`.

This routine is invalid unless the FCode Program has called `fb1-install` and `set-font` and has initialized `frame-buffer-adr` to a valid virtual address.

The cursor position is unchanged, but the cursor character and all characters to the right of the cursor are moved right by *n* places. An error condition exists if an attempt is made to create a line longer than the maximum line size (the value in `#columns`).

fb1-insert-lines

stack: (n --)
code: 178

The built-in default routine to insert *n* blank lines below the cursor on a generic 1-bit-per-pixel framebuffer. This routine is loaded into the `defer` word `insert-lines` by calling `fb1-install`.

This routine is invalid unless the FCode Program has called `fb1-install` and `set-font` and has initialized `frame-buffer-adr` to a valid virtual address.

The cursor position on the screen is unchanged. The cursor line is pushed down, but all lines above it are unchanged. Any lines pushed off of the bottom of the active text area are lost.

fb1-install

stack: (width height #columns #lines --)
code: 17B

This built-in routine installs all of the built-in default routines for driving a generic 1-bit-per-pixel framebuffer. It also initializes most necessary `value`s needed for using these default routines.

`set-font` must be called, and `frame-buffer-adr` initialized, before `fb1-install` is called, because the `char-width` and `char-height` values set by `set-font` are needed when `fb1-install` is executed.

`fb1-install` loads the following `defer` routines with their corresponding `fb1-`(whatever) equivalents: `reset-screen`, `toggle-cursor`, `erase-screen`, `blink-screen`, `invert-screen`, `insert-characters`, `delete-characters`, `insert-lines`, `delete-lines`, `draw-character`, `draw-logo.`

The following `value`s are also initialized:

`screen-width` - set to the value of the passed-in parameter *width* (screen width in pixels)

`screen-height` - set to the value of the passed-in parameter *height* (screen height in pixels)

`#columns` - set to the smaller of the following two: the passed-in parameter *#columns*, and the NVRAM parameter `screen-#columns`

`#lines` - set to the smaller of the following two: the passed-in parameter *#lines*, and the NVRAM parameter `screen-#rows`

`window-top` - set to half of the difference between the total screen height (`screen-height`) and the height of the active text area (`#lines` times `char-height`) `window-left` - set to half of the difference between the total screen width (`screen-width`) and the width of the active text area (`#columns` times `charwidth`), then rounded down to the nearest multiple of 32 (for performance reasons)

Several internal `value`s used by various `fb1-` routine are also set.

fb1-invert-screen

stack: (--)
code: 175

The built-in default routine to invert every visible pixel on a generic 1-bit-per-pixel framebuffer. This routine is loaded into the `defer` word `invert-screen` by calling `fb1-install`.

This routine is invalid unless the FCode Program has called `fb1-install` and has initialized `frame-buffer-adr` to a valid virtual address.

All pixels are inverted (not just the ones in the active text area).

fb1-reset-screen

stack: (--)
code: 171

The built-in default routine to enable a generic 1-bit-per-pixel framebuffer to display data. This routine is loaded into the `defer` word `reset-screen` by calling `fb1-install`. (`reset-screen` is called just after `erase-screen` during the terminal emulator initialization sequence.)

This word is initially a NOP. Typically, an FCode Program will define a hardware-dependent routine to enable video, and then replace this generic function with:

```
: my-video-enable ( -- )  … :

fb1-install
…
['] my-video-enable  to reset-screen
```

fb1-slide-up

stack: (n --)
code: 17C

This is a utility routine. It behaves exactly like `fb1-delete-lines`, except that it doesn't clear the lines at the bottom of the active text area. Its only purpose is to scroll the enable plane for framebuffers that have 1-bit overlay and enable planes.

This routine is invalid unless the FCode Program has called `fb1-install` and `set-font` and has initialized `frame-buffer-adr` to a valid virtual address.

fb1-toggle-cursor

stack: (--)
code: 172

The built-in default routine to toggle the cursor location in a generic 1-bit-per-pixel framebuffer. This routine is loaded into the `defer` word `toggle-cursor` by calling `fb1-install`. The behavior is to invert every pixel in the one-character-size space for the current position of the text cursor.

This routine is invalid unless the FCode Program has called `fb1-install` and `set-font` and has initialized `frame-buffer-adr` to a valid virtual address.

fb8-blink-screen

stack: (--)
code: 184

The built-in default routine to blink or flash the screen momentarily on a generic 8-bit-per-pixel framebuffer. This routine is loaded into the `defer` word `blink-screen` by calling `fb8-install`.

This routine is invalid unless the FCode Program has called `fb8-install` and has initialized `frame-buffer-adr` to a valid virtual address.

This word is implemented simply by calling `fb8-invert-screen` twice. In practice, this can be very slow (several seconds). It is quite common for a framebuffer FCode Program to replace `fb8-blink-screen` with a custom routine that simply disables the video for 20 milliseconds or so. For example:

```
: my-blink-screen  ( -- )  video-off  20 ms  video-on  ;
…
fb8-install
…
['] my-blink-screen   to blink-screen
```

fb8-delete-characters

stack: (n --)
code: 187

The built-in default routine to delete *n* characters to the right of the cursor, on a generic 8-bit-per-pixel framebuffer. This routine is loaded into the `defer` word `delete-characters` by calling `fb8-install`.

This routine is invalid unless the FCode Program has called `fb8-install` and `set-font` and has initialized `frame-buffer-adr` to a valid virtual address.

The cursor position is unchanged. The cursor character and the next n-1 characters to the right of the cursor are deleted, and the remaining characters to the right are moved left by *n* places. The end of the line is filled with blanks.

fb8-delete-lines

stack: (n --)
code: 189

The built-in default routine to delete *n* lines, starting with the line below the cursor line, on a generic 8-bit-per-pixel framebuffer. This routine is loaded into the `defer` word `delete-lines` by calling `fb8-install`.

This routine is invalid unless the FCode Program has called `fb8-install` and `set-font` and has initialized `frame-buffer-adr` to a valid virtual address.

The `n` lines at and below the cursor line are deleted. All lines above the cursor line are unchanged. The cursor position is unchanged. All lines below the deleted lines are scrolled upwards by `n` lines, and `n` blank lines are placed at the bottom of the active text area.

fb8-draw-character

stack: (char --)
code: 180

The built-in default routine for drawing a character on a generic 8-bit-per-pixel framebuffer, at the current cursor location. This routine is loaded into the `defer` word `draw-character` by calling `fb8-install`.

This routine is invalid unless the FCode Program has called `fb8-install` and `set-font` and has initialized `frame-buffer-adr` to a valid virtual address.

If `inverse?` is `true`, then characters are drawn inverted (white-on-black). Otherwise (the normal case) they are drawn black-on-white.

fb8-draw-logo

stack: (line# addr width height --)
code: 18A

The built-in default routine to draw the logo on a generic 8-bit-per-pixel framebuffer. This routine is loaded into the `defer` word `draw-logo` by calling `fb8-install`.

This routine is invalid unless the FCode Program has called `fb8-install` and `set-font` and has initialized `frame-buffer-adr` to a valid virtual address.

See `draw-logo` for more information on the parameters passed.

fb8-erase-screen

stack: (--)
code: 183

The built-in default routine to clear (erase) every pixel in a generic 8-bit-per-pixel framebuffer. This routine is loaded into the `defer` word `erase-screen` by calling `fb8-install`.

This routine is invalid unless the FCode Program has called `fb8-install` and has initialized `frame-buffer-adr` to a valid virtual address.

All pixels are erased (not just the ones in the active text area). If `inverse-screen?` is `true`, then all pixels are set to 0xff, resulting in a black screen. Otherwise (the normal case) all pixels are set to 0, resulting in a white screen.

fb8-insert-characters

stack: (n --)
code: 186

The built-in default routine to insert *n* blank characters to the right of the cursor, on a generic 8-bit-per-pixel framebuffer. This routine is loaded into the `defer` word `insert-characters` by calling `fb8-install`.

This routine is invalid unless the FCode Program has called `fb8-install` and `set-font` and has initialized `frame-buffer-adr` to a valid virtual address.

The cursor position is unchanged, but the cursor character and all characters to the right of the cursor are moved right by *n* places. An error condition exists if an attempt is made to create a line longer than the maximum line size (the value in `#columns`).

fb8-insert-lines

stack: (n --)
code: 188

The built-in default routine to insert *n* blank lines below the cursor on a generic 8-bit-per-pixel framebuffer. This routine is loaded into the `defer` word `insert-lines` by calling `fb8-install`.

This routine is invalid unless the FCode Program has called `fb8-install` and `set-font` and has initialized `frame-buffer-adr` to a valid virtual address.

The cursor position is unchanged. The cursor line is pushed down, but all lines above it are unchanged. Any lines pushed off of the bottom of the active text area are lost.

fb8-install

stack: (width height #columns #lines --)
code: 18B

This built-in routine installs all of the built-in default routines for driving a generic 8-bit-per-pixel framebuffer. It also initializes most necessary `value`s needed for using these default routines.

`set-font` must be called, and `frame-buffer-adr` initialized, before `fb8-install` is called, because the `char-width` and `char-height` values set by `set-font` are needed when `fb8-install` is executed.

`fb8-install` loads the following `defer` routines with their corresponding `fb8-`(whatever) equivalents: `reset-screen`, `toggle-cursor`, `erase-screen`, `blink-screen`, `invert-screen`, `insert-characters`, `delete-characters`, `insert-lines`, `delete-lines`, `draw-character`, `draw-logo`

The following values are also initialized:

`screen-width` - set to the value of the passed-in parameter *width* (screen width in pixels)

`screen-height` - set to the value of the passed-in parameter *height* (screen height in pixels)

`#columns` - set to the smaller of the following two: the passed-in parameter `#columns`, and the NVRAM parameter `screen-#columns`

`#lines` - set to the smaller of the following two: the passed-in parameter *#lines*, and the NVRAM parameter `screen-#rows`

`window-top` - set to half of the difference between the total screen height (`screen-height`) and the height of the active text area (`#lines` times `char-height`)

`window-left` - set to half of the difference between the total screen width (`screen-width`) and the width of the active text area (`#columns` times `char-width`), then rounded down to the nearest multiple of 32 (for performance reasons)

Several internal `value`s are also set that are used by various `fb8-` routines.

fb8-invert-screen

stack: (--)
code: 185

The built-in default routine to XOR (with hex 0xFF) every visible pixel on a generic 8-bit-per-pixel framebuffer. This routine is loaded into the `defer` word `invert-screen` by calling `fb8-install`.

This routine is invalid unless the FCode Program has called `fb8-install` and has initialized `frame-buffer-adr` to a valid virtual address.

All pixels are inverted (not just those in the active text area).

fb8-reset-screen

stack: (--)
code: 181

The built-in default routine to enable a generic 8-bit-per-pixel framebuffer to display data. This routine is loaded into the `defer` word `reset-screen` by calling `fb8-install`. (`reset-screen` is called just after `erase-screen` during the terminal emulator initialization sequence.)

This word is initially a NOP. Typically, an FCode Program will define a hardware-dependent routine to enable video, and then replace this generic function with:

```
: my-video-enable ( -- ) … :
fb8-install
…
['] my-video-enable  to reset-screen
```

fb8-toggle-cursor

stack: (--)
code: 182

The built-in default routine to toggle the cursor location in a generic 8-bit-per-pixel framebuffer. This routine is loaded into the `defer` word `toggle-cursor` by calling `fb8-install`. The behavior is to XOR every pixel with 0xFF in the one-character-size space for the current position of the text cursor.

This routine is invalid unless the FCode Program has called `fb8-install` and `set-font` and has initialized `frame-buffer-adr` to a valid virtual address.

fcode-debug?

stack: (-- save-names?)

This configuration variable is a boolean specifying whether to preserve the names of local FCodes created with named-token in the Forth dictionary. If `fcode-debug?` is *true*, the name fields for FCodes with headers are preserved. If `fcode-debug?` is *false*, discard those names fields.

The suggested default value of `fcode-debug?` is “false”.

fcode-end

stack: (--)
code: none

This tokenizer macro is used to mark the end of an FCode program. `fcode-end` causes the tokenizer to:

- Generate the FCode for `end0`.
- Stop tokenizing the current program.
- Compute the checksum and length for the program and to update the checksum and length fields in the program's FCoder header.

fcode-revision

stack: (-- n)
code: 87

Returns a 32-bit number identifying the version of the device interface. The high 16 bits is the major version number and the low 16 bits is the minor version number. The revision of the device interface described by *IEEE Standard 1275-1994* is "3.0". In a system compatible with that specification, `fcode-revision` will return 0x0003.0000.

For example:

```
: exit-if-not-1275-1994 ( -- )
   fcode-revision h# 30000 <  if  ['] end0 execute  then
;
```

fcode-version1

stack: (--)
code: none

This tokenizer macro is used to start FCode programs intended to be compatible with early OpenBoot systems. That being the case, this macro will seldom be used with PCI devices.

`fcode-version1` generates the FCode header for an FCode program (based upon tokenizer switches). If the default tokenizer switches are used, `fcode-version1` begins the header with the `version1` FCode.

fcode-version2

stack: (--)
code: none

This tokenizer macro causes the tokenizer to:

- Prepare to tokenize subsequent source text.
- Output the FCode for `start1`.
- Output an FCode header.

The length and checksum fields of the FCode header are filled in by the `fcode-end` tokenizer macro.

ferror

stack: (--)
code: FC

Displays an “Unimplemented FCode” error message and stops FCode interpretation at the completion of the function whose evaluation resulted in the execution of `ferror`. All unimplemented FCode numbers resolve to `ferror` in Open Firmware.

The intended use of `ferror` is to determine whether or not a particular FCode is implemented, without checking the FCode version number.

For example:

```
: implemented? ( xt -- flag) ['] ferror <> ;
: my-peer ( prev -- next )
   ['] peer implemented? if
      peer
   else
      ." peer is not implemented" cr
   then
;
```

field

stack: (E: addr -- addr+offset) (offset size “new-name<>” -- offset+size)
code: none
generates: `new-token|named-token|external-token b(field)`

`struct` and `field` are used to create named offset pointers into a structure. For each field in the structure, a name is assigned to the location of that field (as an offset from the beginning of the structure).

The structure being described is:

```
\ size     Bytes  0 - 1
\ flags    Bytes  2 - 5
\ bits     Byte   6
\ key      Byte   7
\ fullname Bytes  8 - 17
\ initials Bytes  8 - 9
\ lastname Bytes  10 - 17
\ age      Bytes  18 - 19
```

The field definitions are shown below. (The numbers in parentheses show the stack *after* each word is created.)

```
struct            ( 0 )
2 field size      ( 2 )   \ equivalent to:   : size      0 + ;
4 field flags     ( 6 )   \ equivalent to:   : flags     2 + ;
1 field bits      ( 7 )   \ equivalent to:   : bits      6 + ;
1 field key       ( 8 )   \ equivalent to:   : key       7 + ;
0 field fullname  ( 8 )   \ equivalent to:   : fullname  8 + ;
2 field initials  ( 10 )  \ equivalent to:   : initials  8 + ;
8 field lastname  ( 18 )  \ equivalent to:   : lastname 10 + ;
2 field age       ( 20 )  \ equivalent to:   : age      18 + ;
constant /record  ( )     \ equivalent to:   20 constant /record
```

Typical usage of these defined words would be:

```
/record buffer: myrecord   \ Create the "myrecord" buffer

myrecord flags l@          \ get flags data
myrecord key   c@          \ get key data
myrecord size  w@          \ get size data

/record                    \ get total size of the array
```

Note that `struct` is primarily a documentation aid that leaves the initial value of the structure's size (i.e. 0) on the stack.

fill

stack: (addr len byte --)
code: 79

Sets `len` bytes of memory beginning at `addr` to the value `byte`. No action is taken if `len` = 0.

find

stack: (pstr -- xt n | pstr 0)
code: none

Finds the command described by the counted string `pstr`. If found, returns -1 (if non-immediate) or +1 (if immediate) on top of the command's execution token. If not found, returns 0 on top of `pstr`.

$find

stack: (name-str name-len -- xt true | name-str name-len false)
code: CB

Takes a string from the stack searches the current search order for it. During normal FCode evaluation, the search order consists of the vocabulary containing the visible methods of the current device node, followed by the Forth vocabulary.

If the word is not found, the original string is left on the stack, with a *false* on top of the stack. If the word is found, the execution token of that word is left on the stack with *true* on top of the stack.

`$find` is an escape hatch, allowing an FCode Program to perform any function that is available in the Open Firmware User Interface but that is not defined as part of the standard FCode interface.

Use `$find` with caution! Different systems or even different versions of Open Firmware may implement different subsets of the User Interface. If your FCode Program depends on a User Interface word, it might not work on some systems.

Example of use:

```
" root-info" $find   ( addr len false | xt true )
if execute           \ if found, then do the function
else ( addr len ) type ." was not found!" cr
then
```

find-device

stack: (dev-str dev-len --)
code: none

Makes the device node specified by *dev-string* the active package.

If *dev-string* is the string "..", the active package is set to the parent of the currently active package. Otherwise, the active package is set using *dev-string* as the device-specifier.

If the specified device is not found, `abort` is executed.

`find-device` is similar to `dev`, except that its argument is a string on the stack instead of text parsed from the input buffer, allowing `find-device` to be used within a definition, with a literal string argument that is compiled into the definition.

For example:

```
" device-alias" find-device
```

See also: `device-end`.

find-method

stack: (method-str method-len phandle -- false | xt true)
code: 207

Locates the method named by *method-str method-len* within the package *phandle*. Returns *false* if the package has no such method, or *xt* and *true* if the operation succeeds. Subsequently, *xt* can be used with `call-package`.

For example:

```
: tftp-load-avail? ( -- exist? )
   " obp-tftp" find-package  if  ( phandle )
      " load"  rot find-method if ( xt )
         drop true exit
      then
   then
   false
;
```

find-package

stack: (name-str name-len -- false | phandle true)
code: 204

Locates a package whose name is given by the string *name-str name-len*. If the package can be located, returns its *phandle* and *true*. Otherwise returns *false*.

The name is interpreted relative to the `/packages` device node. For example, if *name-str name-len* represents the string `"disk-label"`, the package in the device tree at “`/packages/disk-label`” will be located.

If there are multiple packages with the same name (within the `/packages` node), the *phandle* for the most recently created one is returned.

For example:

```
: tftp-load-avail? ( -- exist? )
   " obp-tftp" find-package  if  ( phandle )
      " load"  rot find-method if ( xt )
         drop true exit
      then
   then
   false
;
```

finish-device

stack: (--)
code: 127

The two words `finish-device` and `new-device` let a single FCode Program declare more than one entry into the device tree. This capability is useful when a single PCI card contains two or more essentially independent devices, to be controlled by two or more separate operating system device drivers.

Typical usage:

```
fcode-version2  \ begin a new device tree entry
…driver#1…
finish-device   \ terminate device tree entry#1
new-device      \ begin a new device tree entry
…driver#2
finish-device   \ terminate device tree entry#2
new-device      \ begin a new device tree entry
…driver#3…
fcode-end     \ terminate device tree entry#3
```

There is an implicit `new-device` call at the beginning of an FCode Program (at `version1` or `start1`), and an implicit `finish-device` call at the end of an FCode Program (at `end0`). Thus, FCode Programs that only define a single device and driver will never need to call `finish-device` or `new-device`.

fload

stack: ([filename<cr>] --)
code: none

This command allows FCode text programs to be broken into function blocks for better clarity, portability and re-use. It behaves similarly to the `#include` statement in the C language. Arbitrary nesting of files with `fload` is allowed i.e. an `fload`'d file may include `fload` commands.

When `fload` is encountered, the Tokenizer continues tokenizing the FCode found in the file *filename*. When the file *filename* has been tokenized, tokenizing resumes on the file that called *filename* with `fload`.

For example:

```
fload my-disk-package.fth
```

Note – `fload` commands won't work when downloading text in source-code form using `dl`.

There are several ways to overcome this problem:

- Manually merge the files into one larger text file and download the merged file with `dl.`
- Create a "load file" and use the load file in conjunction with `dl`. (See "Downloading Multiple Files with dl and fload" on page 28 for a detailed explanation of this technique.)
- Tokenize the files first and then download and execute the FCode in binary form.

fm/mod

stack: (d n -- rem quot)
code: none

Divides *d* by *n* and returns *rem* and *quot*.

>font

stack: (char -- addr)
code: 16E

This routine converts a character value (ASCII 0-0xFF) into the address of the font table entry for that character. For the normal, built-in font, only ASCII values 0x21-0x7E result in a printable character, other values will be mapped to a font entry for "blank".

This word is only of interest if you are implementing your own character-drawing routines.

Note – `>font` will generate invalid results unless `set-font` has been called to initialize the font table to be used.

---

fontbytes

stack: (-- bytes)
code: 16F

A `value`, containing the interval between successive entries in the font table. Each entry contains the next scan line bits for the desired character. Each scan line is normally 12 pixels wide, and is stored as one bit per pixel, thus taking 1 1/2 bytes per scan line. The standard value for `fontbytes` is 2, meaning that the next scan line entry is 2 bytes after the previous one (the last 1/2 byte is wasted space).

This word *must* be set to the appropriate value if you wish to use *any* `fb1-` or `fb8-` utility routines or `>font`. This can be done with `to`, but is normally done by calling `set-font`.

The standard value for `fontbytes` is one of the parameters returned by `default-font`.

forth

stack: (--)
code: none

Make Forth the context vocabulary.

frame-buffer-adr

stack: (-- addr)
code: 162

This `value` returns the virtual address of the beginning of the current framebuffer memory. It *must* be set to an appropriate virtual address (using `to`) in order to use *any* of the `fb1-` or `fb8-` utility routines. It is suggested that this same `value` variable be used in any of your custom routines that require a framebuffer address, although of course you are free to create and use your own variable if you wish.

Generally, you should only map in the framebuffer memory just before you are ready to use it, and unmap it if it is no longer needed. Typically, this means you should do your mapping in your "install" routine, and unmap it in your "remove" routine (see `is-install` and `is-remove`). Here's some sample code:

```
h# 2.0000  constant  /frame    \ # of bytes in frame buffer
h# 40.0000 constant  foffset   \ Location of frame buffer

: video-map  ( -- )
   my-address  foffset +  /frame  map-pci  to  frame-buffer-adr
;
: video-unmap  ( -- )
   frame-buffer-adr  /frame  free-virtual
   -1  to  frame-buffer-adr    \ Flag accidental accesses to a
                               \ now-illegal address
;

: power-on-selftest  ( -- )
   video-map
   ( test video memory )
   video-unmap
;
power-on-selftest

: my-install  ( -- )
   video-map
   …
;
: my-remove   ( -- )
   video-unmap
   …
;
…
['] my-install is-install
['] my-remove is-remove
```

Note – United States Patent No. 4,633,466, "Self Testing Data Processing System with Processor Independent Test Program", issued December 30, 1986 may apply to some or all elements of Open Firmware selftest. Anyone implementing Open Firmware should take such steps as may be necessary to avoid infringement of that patent and any other applicable intellectual property rights.Consequently, the above example selftest is only intended to illustrate the concept of mapping a resource immediately before use, and of unmapping a resource immediately after use.

---

free-mem

stack: (a-addr len --)
code: 8C

Frees up *len* memory allocated by `alloc-mem`. The arguments *a-addr* and *len* must be the same as those used in a previous `alloc-mem` command.

For example:

```
0 value my-string           \ Holds address of temporary
: .upc-string ( addr len -- ) \ convert to uppercase and print.
   dup alloc-mem to my-string                    ( addr len )
   tuck my-string swap move                      ( len )
   my-string over bounds ?do i c@ upc i c! loop  ( len )
   my-string over type                           ( len )
   my-string swap free-mem
;
```

free-virtual

stack: (virt size --)
code: 105

Destroys an existing mapping and any `"address"` property.

If the package associated with the current instance has an `"address"` property whose first value encodes the same address as *virt*, delete that property. In any case, execute the parent instance's `map-out` method with *virt size* as its arguments.

.fregisters

stack: (--)
code: none

Displays floating-point registers (if present). The exact set of registers displayed, and the format, is system-dependent.

get-inherited-property

stack: (name-str name-len -- true | prop-addr prop-len false)
code: 21d

Locates, within the package associated with the current instance or any of its parents, the property whose name is *name-addr name-len*. If the property exists, returns the property value array *prop-addr prop-len* and *false*. Otherwise returns *true*.

The order in which packages are searched is the current instance first, followed by its immediate parent, followed by its parent's parent, and so on. This is useful for properties with default values established by a parent node, with the possibility of a particular child node “overriding” the default value.

For example:

```
: clock-frequency ( -- val.addr len false | true  )
   " clock-frequency" get-inherited-property
;
```

get-msecs

stack: (-- n)
code: 125

Returns the current value in a free-running system counter. The number returned is a running total, expressed in milliseconds. You can use this for measuring time intervals (by comparing the starting value with the ending value). No assumptions should be made regarding the absolute number returned; only relative interval comparisons are valid.

No assumptions should be made regarding the *precision* of the number returned. In some systems, the value is derived from the system clock, which typically ticks once per second. Thus, the value returned by `get-msecs` on such a system will be seen to increase in jumps of 1000 (decimal), once per second.

For a delay timer of millisecond accuracy, see `ms`.

get-my-property

stack: (name-str name-len -- true | prop-addr prop-len false)
code: 21A

Locates, within the package associated with the current instance, the property named by *name-addr name-len*. If the property exists, returns the property value array *val-addr val-len* and *false*. Otherwise returns *true*.

For example:

```
: show-model-name ( -- )
    " model" get-my-property if  ( val.addr len )
        ." model  property is missing "
    else  ( )
        ." model name is " type
    then ( ) cr
;
```

get-package-property

stack: (name-str name-len phandle -- true | prop-addr prop-len false)
code: 21F

Locates, within the package *phandle*, the property named by *name-addr name-len*. If the property exists, returns the property value array *prop-addr prop-len* and *false*. Otherwise returns *true*.

For example:

```
: show-model-name ( -- )
   my-self ihandle>phandle ( phandle )
   " model" rot get-package-property 0= if  ( val.addr len )
      ." model name is " type cr
   else  ( )
      ." model  property is missing " cr
   then ( )
;
```

get-token

stack: (fcode# -- xt immediate?)
code: DA

Returns the execution token *xt* of the word associated with FCode number *fcode#* and a flag *immediate?* that is `true` if and only if that word will be executed (rather than compiled) when the FCode Evaluator encounters its FCode number while in compilation state.

go

stack: (--)
code: none

Executes or resumes execution of a program in memory by restoring the processor state from the `saved-program-state` memory area and beginning/resuming the execution of the machine-code program.

Resume execution at the address saved in the `saved-program-state` program counter register. This will normally contain the initial value for a newly-loaded program, or the resumption address for a suspended program. However, the saved program counter register can be altered by the user, causing the program to resume (when `go` is executed) from an arbitrary address.

This command has no effect unless `state-valid` contains `true`.

`go` can be used in conjunction with other commands in one of several ways:

- After `load` (which also initializes `saved-program-state`), `go` executes the program just downloaded.
- After a program is suspended by entering the implementation-dependent "abort-sequence" (which saves the processor state in `saved-program-state`), `go` resumes execution of the suspended program.
- When testing a program with breakpoints, and after a breakpoint has been encountered (which saves the processor state in `saved-program-state`), `go` resumes execution of the program being tested.

gos

stack: (n --)
code: none

Executes `go` *n* times.

h# “aych number”

stack: ([number< >] -- n)
code: none
generates: b(lit) xx-byte xx-byte xx-byte xx-byte

Causes the compiler/interpreter to interpret the immediately following number as a hexadecimal number (base sixteen), regardless of any previous settings of hex, decimal or octal. Only the immediately following number is affected. The value of base is unchanged.

For example:

```
decimal
h# 100 ( equals decimal 256 )
100    ( equals decimal 100 )
```

See also: d#, o#.

.h “dot aych”

stack: (n --)
code: none
generates: base @ swap d# 16 base ! . base !

Displays *n* in hex (using .) The value of base is not permanently affected.

headerless

stack: (--)
code: none

Causes all subsequent definitions to be created in FCode without the name field (the “head”). (See named-token and new-token.) This is sometimes done to save space in the final FCode ROM, or possibly to make it more difficult to reverse-engineer an FCode Program.

All such headerless words can be used normally within the FCode Program, but cannot be called interactively from the User Interface for testing and development purposes.

Unless ROM space and/or dictionary space is a major consideration, try not using headerless words, because they make debugging more difficult.

headerless remains in effect until headers or external is encountered.

For example:

```
headerless
h# 3 constant reset-scsi
```

headers

stack: (--)
code: none

Causes all subsequent definitions to be saved with the name field (the “head”) intact. This is the initial default behavior.

Note that even normal FCode words (with heads) cannot be called interactively from the User Interface unless the NVRAM parameter `fcode-debug?` has been set to `true` before a system reset.

`headers` remains in effect until `headerless` or `external` is encountered.

For example:

```
headers
: cnt@  ( -- w )
   transfer-count-lo rb@
   transfer-count-hi rb@
   bwjoin
;
```

"height"

This standard property is associated with `display` devices. The property value is an integer (encoded with `encode-int`) that specifies the number of displayable pixels in the “y” dimension of the display.

See also: `property`

help

stack: ("{name}<eol>" --)

Provides information for the specified category or command.

If *name* is a specific command, lists help for that command, if available. Otherwise, displays an implementation-dependent message. For example:

```
ok help command-name
```

If *name* is a category, lists all help messages for commands in that category, or a list of sub-categories. For example:

```
ok help category-name
```

If *name* is omitted, general help and a list of available categories is provided. The number and names of categories/subcategories are implementation dependent.

here

stack: (-- addr)
code: AD

`here` returns the address of the next available dictionary location.

hex

stack: (--)
code: none
generates: `b(lit) 16 base !`

If used outside of a definition, commands the tokenizer program to interpret subsequent numbers in hex (base 16). If used within a definition, changes the value in base affecting later numeric output when the FCode Program is executed.

See also: base

hold

stack: (char --)
code: 95

Inserts *char* into a pictured numeric output string. Typically used between <# and #>.

For example:

```
: .32 ( n -- )
   base @ >r hex
   <# # # # #  ascii . hold # # # # #> type
   r> base !
   space
;
```

hop

stack: (--)
code: none

hop is one of the breakpoint commands. After a breakpoint has been encountered, hop executes a single instruction, or an entire subroutine call.

hop behaves similarly to step except that, if the instruction to be executed is a subroutine call, hop executes the entire subroutine before stopping instead of just the call instruction.

If the execution of that subroutine results in encountering another breakpoint, the result is implementation-dependent.

hops

stack: (n --)
code: none

Execute hop *n* times.

i

stack: (-- index) (R: loop-sys -- loop-sys)
code: 19

index is a copy of the loop index of the immediately-enclosing do or ?do loop. Indeterminate results will be obtained if i is used elsewhere.

For example:

```
: simple-loop  ( start len -- )
   bounds ?do i .h cr loop
;
```

if

stack: (C: -- orig-sys)
(do-next? --)
code: none
generates: `b?branch +offset`

Execute the immediately-following code if *do-next*? is true. Used in the form:

```
do-next? if…else…then
```

or

```
do-next? if…then
```

If *do-next*? is true, the words following `if` are executed and the words following `else` are skipped. The `else` part is optional. If *do-next*? is false, words from `if` through `else`, or from `if` through `then` (when no `else` is used), are skipped.

ihandle>phandle

stack: (ihandle -- phandle)
code: 20B

Returns the *phandle* of the package from which the instance *ihandle* was created. This is often used with `get-package-property` to read the properties of the package corresponding to a given *ihandle*.

For example:

```
: show-parent ( -- )
   my-parent ihandle>phandle " name" rot
   get-package-property 0= if
     ." my-parent is " type cr
   then
;
```

immediate

stack: (--)
code: none

Declares the previous definition as “immediate”.

>in

stack: (-- a-addr)
code: none

A `variable` containing the offset of the next input buffer character.

init-program

stack: (--)
code: none

Initializes `saved-program-state` to the system-dependent initial program state required for beginning the execution of a client program.

input

stack: (dev-str dev-len --)
code: none

Selects the specified device for console input by searching for a device node matching the pathname or device-specifier given by *dev-str dev-len*.

- If such a device is found, search for its `read` method.
- If the read method is found, open the device, as with `open-dev`.
- If the open succeeds, execute the device's `install-abort` method, if any.
- If any of these steps fails, display an appropriate error message and return without performing the steps following the one that failed.

If there is a console input device, as indicated by a nonzero value in the `stdin` variable, execute the console input device's `remove-abort` method and close the console input device. Set `stdin` to the ihandle of the newly opened device, making it the new console input device.

For example:

```
ok " device-alias" input
```

input-device

stack: (-- dev-str dev-len)

The value of this configuration variable is a string describing the device-specifier of the device to be established as the default console input device by `install-console`.

The suggested default value is "keyboard".

For example:

```
ok setenv input-device device-alias <eol>
```

insert-characters

stack: (n --)
code: 15D

`insert-characters` is one of the `defer` words of the display device interface. The terminal emulator package executes `insert-characters` when it has processed a character sequence that calls for opening space for characters to the right of the cursor. Without moving the cursor, `insert-characters` moves the remainder of the line to the right, thus losing the *n* rightmost characters in the line, and fills the *n* vacated character positions with the background color.

This word is initially empty, but *must* be loaded with an appropriate routine in order for the terminal emulator to function correctly. This can be done with `to`, or it can be loaded automatically with `fb1-install` or `fb8-install` (which loads `fb1-insert-characters` or `fb8-insert-characters`, respectively).

insert-lines

stack: (n --)
code: 15F

`insert-lines` is one of the defer words of the display device interface. The terminal emulator package executes `insert-lines` when it has processed a character sequence that calls for opening space for lines of text below the cursor. Without moving the cursor, `insert-lines` moves the cursor line and all following lines down, thus losing the *n* bottom lines. and fills the *n* vacated lines with the background color.

This word is initially empty, but *must* be loaded with an appropriate routine in order for the terminal emulator to function correctly. This can be done with `to`, or it can be loaded automatically with `fb1-install` or `fb8-install` (which load `fb1-insert-lines` or `fb8-insert-lines`, respectively).

install-abort

stack: (--)
code: none

Instructs the device driver to begin periodic polling for a keyboard abort sequence. If a keyboard abort sequence is subsequently encountered, `abort` is executed.

This command is executed when the device is selected as the console input device.

install-console

stack: (--)
code: none

Activates the console function and selects the input and output devices as follows:

a) Activate the console so that subsequent input (e.g. `key`) and output (e.g. `emit`) will use the devices selected by `input` and `output`.

b) Execute output with the value returned by `output-device`.

c) Execute input with the value returned by `input-device`.

d) If the above code failed and there is a fallback device to be used for console functions, select that device as the console device.

`install-console` may take other system-dependent actions to insure that a console device is available in the event of a failure, and may display messages indicating that such action has been taken.

instance

stack: (--)
code: C0

Modifies the next occurrence of `value`, `variable`, `defer` or `buffer:` to create instance-specific data instead of static data. Re-allocates the data each time a new instance of this package is created.

For example:

```
-1 instance value my-chip-reg
```

.instruction

stack: (--)
code: none

Displays the address where the last breakpoint occurred and the instruction that would have executed next if the breakpoint had not been there. The instruction-display format is system-specific.

interpose

stack: (addr len phandle --)
code: 12B

Schedule the package identified by *phandle*' for interposition, with the string *addr len* as its arguments.

If a package is currently scheduled for interposition when `interpose` is executed, the result is undefined (i.e. an Open Firmware implementation need not support multiple simultaneous interposition attempts).

Note – This function must be executed only during the creation of an instance chain (i.e. during the execution of a package's `open` method during pathname resolution in `open-dev` context, as in clauses (f2), (k1iii) and (m2) of section 4.3.1 of *IEEE Standard 1275-1994*.

"interrupts"

This property specifies the interrupt level(s) used by this device and possibly other appropriate information (such as interrupt vectors). The level given is the bus-specific (local) level, not the CPU level. (The operating system driver translates the local level to the system level. This enables the FCode driver to be portable across platforms. See “"interrupts"” on page 73.) The actual format of the data is bus-specific; see the appropriate 1275 machine-specific binding document for details.

The property value is an arbitrary number of (bus-specific) interrupt specifiers each typically encoded with `encode-int`.

See also: `"interrupts"` in Chapter 5 “Properties”

"intr"

This property specifies SBus interrupt level(s) and vector(s) used by this device.

This property is included in this glossary because of the possibility that, even on systems that nominally do not support SBus, SBus devices might be used via a bus-to-bus bridge.

For complete details, see *IEEE Standard 1275-1994*.

inverse?

stack: (-- white-on-black?)
code: 154

This `value` is part of the display device interface. The terminal emulator package sets `inverse?` to `true` when the escape sequences that it has processed have indicated that subsequent characters are to be shown with foreground and background colors exchanged, and to `false`, indicating normal foreground and background colors, otherwise.

The `fb1-` and `fb8-` frame buffer support packages draw characters with foreground and background colors exchanged if `inverse?` is `true`, and with normal foreground and background colors if `inverse?` is `false`.

`inverse?` affects the character display operations `draw-character`, `insert-characters`, and `delete-characters`, but not the other operations such as `insert-lines`, `delete-lines` and `erase-screen`.

inverse-screen? should be monitored as needed if your FCode Program is implementing its own set of framebuffer primitives.

See also: `inverse-screen?`

inverse-screen?

stack: (-- black?)
code: 155

This `value` is part of the display device interface. The terminal emulator package sets `inverse-screen?` to `true` when the escape sequences that it has processed have indicated that the foreground and background colors are to be exchanged for operations that affect the background, and to `false`, indicating normal foreground and background colors, otherwise.

The `fb1-` and `fb8-` frame buffer support packages perform screen drawing operations other than character drawing operations with foreground and background colors exchanged if `inverse-screen?` is `true`, and with normal foreground and background colors is false.

`inverse-screen?` affects background operations such as `insert-lines`, `delete-lines` and `erase-screen`, but not character display operations such as `draw-character`, `insert-characters` and `delete-characters`.

When `inverse-screen?` and `inverse?` are both true, the colors are exchanged over the entire screen, and subsequent characters are not highlighted with respect to the (inverse) background. For exchanged screen colors and highlighted characters, the setting are `inverse-screen? true` and `inverse? false`.

inverse-screen? should be monitored as needed if your FCode Program is implementing its own set of framebuffer primitives.

invert

stack: (x1 -- x2)
code: 26

x2 is the one's complement of *x1* i.e. all the one bits in *x1* are changed to zero, and all the zero bits are changed to one.

For example:

```
: clear-lastbit ( -- )
   my-reg rl@ 1 not and my-reg rl!
;
```

See also `0=`.

invert-screen

stack: (--)
code: 15C

`invert-screen` is one of the `defer` words of the display device interface. The terminal emulator package executes `invert-screen` when it has processed a character sequence that calls for exchanging the foreground and background colors (e.g. changing from black-on-white to white-on-black).

`invert-screen` changes all pixels on the screen so that pixels of the foreground color are given the background color, and vice versa, leaving the colors that will be used by subsequent text output unaffected.

This word is initially empty, but *must* be loaded with an appropriate routine in order for the terminal emulator to function correctly. This can be done with `to`, or it can be loaded automatically with `fb1-install` or `fb8-install` (which load `fb1-invert-screen` or `fb8-invert-screen`, respectively).

io

stack: (dev-str dev-len --)
code: none

Selects the indicated device for console input and output by executing `input` followed by `output` with *dev-str dev-len* as arguments in both cases. For example:

```
ok " device-alias" io
```

is-install

stack: (xt --)
code: 11C

Creates `open`, `write`, `draw-logo` and `restore` methods for display devices. *xt* is the execution token of a routine whose stack diagram is (--), and that initializes the display device.

For any PCI framebuffer that is to be used by the boot ROM before or during booting, `is-install` declares the FCode procedure that should be used to install (i.e. initialize) that framebuffer. Note that this is distinct from any once-only power-on initialization that should be performed during the probing process itself.

The `is-install` routine and `is-remove` routine should comprise a matched pair that may be performed alternately as many times as needed. Typically, the `is-install` routine performs mapping functions, enables PCI memory and/or I/O space accesses and performs some initialization. Typically, the `is-remove` performs any cleanup functions and then does a complementary disabling of the appropriate address space(s) and unmaps the existing mappings.

A partial, typical code example follows:

```
fcode-version2
…
: map-devices ( -- )        \ Map register and buffer
   map-register
   map-buffer
;
…
: install-me ( -- )         \ Do this to start using this device
   map-devices
   initialize-devices
   fb8-install              \ Install default defer word behaviors
;
: remove-me ( -- )          \ Do this to stop using this device
   reset-buffers
   unmap-devices
;
…
['] install-me is-install   \ Declare "install" routine
['] remove-me  is-remove    \ Declare "remove" routine
['] test-me    is-selftest  \ Declare "selftest" routine

fcode-end
```

See also: “is-install Actions” on page 140

is-remove

stack: (xt --)
code: 11D

Creates a `close` method for display devices that should de-allocate a framebuffer that is no longer going to be used. Typical de-allocation would include unmapping memory and clearing buffers. For example:

```
fcode-version2
…
: remove-me   ( -- )    \ Do this to stop using this device
   reset-buffers
   unmap-devices
;
…
['] install-me  is-install   \ Declare "install" routine
['] remove-me   is-remove    \ Declare "remove" routinea
['] test-me     is-selftest  \ Declare "selftest" routine

fcode-end
```

The routine loaded with `is-remove` should form a matched pair with the routine loaded with `is-install`. See `is-install` for more details.

is-selftest

stack: (xt --)
code: 11E

Creates a `selftest` method for display devices that will perform a self test of the framebuffer. For example:

```
fcode-version2
…
: test-me   ( -- fail? )   \ self test method
   …
;
…
['] install-me  is-install   \ Declare "install" routine
['] remove-me   is-remove    \ Declare "remove" routine
['] test-me     is-selftest  \ Declare "selftest" routine

fcode-end
```

This declaration is typically performed in the same place in the code as `is-install` and `is-remove`.

The self test routine should return a status parameter on the stack indicating the results of the test. A zero value indicates that the test passed. Any nonzero value indicates that the self test failed, but the actual meaning for any nonzero value is not specified. (`memory-test-suite` returns a flag meeting these specifications.)

`selftest` is not automatically executed. For automatic testing, devices should perform a quick sanity check as part of the `install` routine. See "selftest (-- error#)" on page 53.

(is-user-word)

stack: (E: … -- ???) (name-str name-len xt --)
code: 214

Creates a Forth word (not a package method) whose name is given by *name-str name-len* and whose behavior is given by the execution token *xt* which must refer to a static method. This allows an FCode Program to define new User Interface commands.

For example:

```
" xyz-abort" ' my-abort (is-user-word)
```

j

stack: (-- index) (R: sys -- sys)
code: 1A

index is a copy of the loop index of the next outer `do` or `?do` loop. Indeterminate results will be obtained if `i` is used elsewhere. For example:

```
10 0 do
   …
   33 20 do
      …
      j \ Returns a value in the range 0 to 9
      …
   loop
  …
loop
```

Usually, `do` loops should not be nested this deeply inside a single definition. Forth programs are generally more readable if inner loops are defined inside a separate word.

key

stack: (-- char)
code: 8E

A `defer` word that reads the next ASCII character from the keyboard. If no character has been typed since `key` or `expect` was last executed, `key` waits until a new character is typed. All valid ASCII characters can be received. Control characters are not processed by the system for any editing purpose. Characters received by `key` are not displayed.

For example:

```
: continue? ( -- continue? )
   ." Want to Continue? Enter Y/N" key dup emit
   dup ascii Y = ascii y rot = or
;
```

See also: `key?`

key? “key question”

stack: (-- pressed?)
code: 8D

A `defer` word returning *true* if a character has been typed on the keyboard since the last time that `key` or `expect` was executed. The keyboard character is not consumed.

Use `key?` to make simple, interruptible infinite loops:

```
begin … key? until
```

The contents of the loop will repeat indefinitely until any key is pressed.

See also: `key`

keyboard

The suggested default value for the `input-device` configuration variable.

l! “el store”

stack: (quad qaddr --)
code: 73

The 32-bit value *quad* is stored at location *qaddr*. *qaddr* must be 32-bit aligned.

See also: `rl!`

l, “el comma”

stack: (quad --)
code: D2

Compile a 32-bit number into the dictionary. The dictionary pointer must be 2-byte-aligned.

For example:

```
\ to create an array containing integers 40004000 23 45 6734
create my-array 40004000 l, 23 l, 45 l, 6734 l,
```

l@ “el fetch”

stack: (qaddr -- quad)
code: 6E

Fetch the 32-bit number stored at *qaddr*. *qaddr* must be 32-bit aligned.

See also: `rl@`

/l “per el”

stack: (-- n)
code: 5C

n is the number of address units to a 32-bit word, typically 4.

/l*

stack: (nu1 -- nu2)
code: 68

nu2 is the result of multiplying *nu1* by `/l`. This is the portable way to convert an index into a byte offset.

<l@

stack: (qaddr -- n)
code: 242

Fetch quadlet from qaddr, sign-extended.

This function is only available on 64-bit implementations.

la+

stack: (addr1 index -- addr2)
code: 60

Increments *addr1* by *index* times the value of `/l`. This is the portable way to increment an address.

la1+

stack: (addr1 -- addr2)
code: 64

Increments *addr1* by the value of `/l`. This is the portable way to increment an address.

label

stack: (E: -- addr)
(C: "new-name< >" -- code-sys)
code: none

Begins creation of a machine-code sequence called *new-name*. Interprets the following commands as assembler mnemonics.

Commands created by `label` leave the address of the code on the stack when executed.

As with `code`, `label` is present even if the assembler is not installed. In this case, machine-code must be entered into the dictionary explicitly by value i.e. with: `c,`, `w,`, `l,` or `,`. The machine-code sequence is terminated by the `c;` or `end-code` commands.

For example:

```
ok label new-name
ok … assembler mnemonics …
ok end-code
```

Later used as:

```
  new-name  ( machine-code-addr )
```

code-sys is balanced by the corresponding `c;` or `end-code`.

lbflip

stack: (quad1 -- quad2)
code: 227

Reverse the bytes within a 32-bit datum.

lbflips

stack: (qaddr len --)
code: 228

Reverse the bytes within each 32-bit datum in the given region.

The region begins at *qaddr* and spans *len* bytes. The behavior is undefined if *len* is not a multiple of `/l`.

lbsplit

stack: (quad -- byte1.lo byte2 byte3 byte4.hi)
code: 7E

Splits a 32-bit datum into four bytes. All but the least significant 8 bits of each stack result are zero.

lcc

stack: (char1 -- char2)
code: 82

char2 is the lower case version of *char1*. If *char1* is not an upper case letter, it is unchanged. For example:

```
ok ascii M lcc emit
m
ok
```

See also: `upc`

leave

stack: (--) (R: loop-sys --)
code: none
generates: `b(leave)`

May only be used within a `do` or `?do` loop. Transfers execution to just past the next `loop` or `+loop`. The loop is terminated and loop control parameters are discarded.

`leave` may appear within other control structures that are nested within the `do` loop structure. More than one `leave` may appear within a `do` loop.

To leave the word containing the `do` or `?do` loop (not just the loop itself), use the phrase `unloop exit` instead of `leave`.

For example:

```
: search-pat ( pat addr len -- found? )
   rot false swap 2swap  ( false pat addr len )
   bounds ?do  ( flag pat )
      i @ over  =  if  drop true swap leave  then
   loop
   drop
;
```

See also: `exit`, `unloop`

?leave

stack: (exit? --) (R: sys --)
code: none
generates: `if leave then`

If *exit*? is *true* (nonzero), `?leave` transfers control to just beyond the next `loop` or `+loop`. The loop is terminated and loop control parameters are discarded. If *exit*? is zero, no action is taken. May only be used within a `do` or `?do` loop.

`?leave` may appear within other control structures that are nested within the `do` loop structure. More than one `?leave` may appear within a `do` loop.

For example:

```
: show-mem ( vaddr -- )  \ display h# 10 bytes
   dup h# 9 u.r 5 spaces h# 10 bounds  do  i c@ 3 u.r  loop
;
: .mem ( vaddr size -- )
   bounds  ?do  i show-mem key? ?leave  h# 10  +loop
;
```

left-parse-string

stack: (str len char -- R-str R-len L-str L-len)
code: 240

Splits the input string at the first occurrence of delimiter *char*. For example:

```
" test;in;g" ascii ; left-parse-string
```

would leave the address and length of two strings on the stack:

“`in;g`” and “`test`”.

The delimiter character may be any ASCII character. Note that if the delimiter is not found within the string, the effect is as if the delimiter was found at the very end. For example:

```
" testing" ascii q left-parse-string
```

would leave on the stack “” and “`testing`”.

line#

stack: (-- line#)
code: 152

A `value`, set and controlled by the terminal emulator, that contains the current cursor line number. A value of 0 represents the topmost line of available text space — *not* the topmost pixel of the framebuffer.

This word should be monitored as needed if your FCode Program is implementing its own set of framebuffer primitives.

For example:

```
: set-line  ( line -- ) 0 max  #lines 1- min  to line# ;
```

See also: `window-top`

#line

stack: (-- a-addr)
code: 94

A `variable` containing the number of output lines since the last user interaction i.e. since the last ok prompt. `#line` is incremented whenever `cr` executes. The value in this `variable` is used to determine when to pause during long display output, such as `dump`. Its value is reset each time the ok prompt displays.

See also: `exit?`

"linebytes"

This standard property is associated with `display` devices. The property value is an integer (encoded with `encode-int`) that specifies the number of pixels in a single scan line of the display.

See also: `property`

linefeed

stack: (-- 0x0A)
code: none
generates: `b(lit) 00 00 00 0x0A`

Leaves the ASCII code for the linefeed character (i.e. Control-J) on the stack.

#lines

stack: (-- rows)
code: 150

`#lines` is a `value` that is part of the display device interface. The terminal emulator package uses it to determine the height (number of rows of characters) of the text region that it manages. The `fb1-` and `fb8-` frame buffer support packages also use it for a similar purpose.

The value of `#lines` must be set to the desired height of the text region. This can be done with `to`, or it can be handled automatically as one of the functions performed by `fb1-install` or `fb8-install`. The value set by `fbx-install` is the smaller of the passed `#lines` parameter and the `screen-#rows` NVRAM parameter.

For example:

```
: set-line  ( line -- ) 0 max  #lines 1- min  to line# ;
```

literal

stack: (C: x1 --)
(-- x1)
code: none

Compiles a number. When later executed, leaves the number on the stack.

load

stack: ("{device-specifier< >} {arguments}<eol>" --)
code: none

The User Interface provides a `load` method which can, in turn, select a source device and use that device's `load` method to load the specified program into memory. If the *device-specifier* and / or *arguments* are not provided to `load` on the command line, `load` uses defaults as described below. The parsing, loading and default argument selection processes are described below.

Parameter Parsing:

`load` finds the first, space-delimited argument, *first-arg*.

- If *first-arg* is the empty string, `load` sets *device-specifier* to the default device and *arguments* to the default arguments as specified below, and proceeds with the loading process as specified below.
- If *first-arg* begins with the "`/`" character, or if it is the name of a defined devalias, `load` sets *device-specifier* to *first-arg*. `load` then skips leading space delimiters and sets *arguments* to the remainder of the command line.
- Otherwise, `load` sets *device-specifier* to the default device and `arguments` to the portion of the command line beginning at *first-arg* and continuing to the end of the line (including *first-arg* itself).

Loading Process:

If the client interface is implemented, `load` saves `arguments` and the device-path corresponding to *device-specifier* so they may be retrieved later via the client interface.

Using `open-dev`, `load` opens the package specified by *device-specifier*, thus obtaining an ihandle. If unsuccessful, `load` executes the equivalent of `abort`, thus stopping the loading process. Otherwise `load` uses `$call-method` to execute the `load` method of that ihandle, passing the system-dependent default load address to that `load` method as its argument. `load` then uses `close-dev` to close that ihandle.

If the device's `load` method succeeds, and the beginning of the loaded image is a valid client program header for the system, `load` allocates memory at the address and of the size specified in that header, moves the loaded image to the address, and performs the function of `init-program` to initialize `saved-program-state` so that a subsequent `go` command will begin execution of that program.

Default Device and Default Arguments:

The default arguments are given by the value of `boot-file` if `diagnostic-mode`? is `false`. Otherwise the default arguments are given by the value of `diag-file`.

The default device is given by the value of `boot-device` if `diagnostic-mode?` is `false`. Otherwise the default device is given by the value of diag-device.

Either `boot-device` or `diag-device` may contain a list of device-specifiers separated by spaces. If that list contains only one entry, that entry is the default device. If that list contains more than one entry, the system attempts to open, as with `open-dev`, each specified device in turn, beginning with the first entry in the list and proceeding to the next-to-last entry. If an open succeeds, the device is closed, as with `close-dev`, and that device-specifier becomes the default device (it will be subsequently opened again by the loading process). If the last entry is reached without any prior successful opens, the last entry becomes the default device, without having been opened as part of the default device selection process.

For example:

```
ok load device-specifier arguments
```

See also: `boot`

load

stack: (addr -- len)
code: none

A device's `load` method can be used to load a client program from the device into memory beginning at address *addr*. `load` returns *len*, the size in bytes of the program that was loaded. The package containing `load` must be open before `load` is executed.

If the device can contain several such programs, the instance-arguments (as returned by `my-args`) can be used in a device-dependent manner to select the particular program.

load-base

stack: (-- addr)

This platform-specific configuration variable is an integer specifying the default load address for client programs when using the `load` method. The default value is implementation dependent.

"local-mac-address"

This property specifies the 48-bit IEEE 802.3-style MAC address assigned to the device represented by the package, of device type `"network"`, containing this property. The absence of this property indicates that the device does not have a permanently-assigned MAC address.

The property value is an array of six bytes encoded with `encode-bytes`.

For example:

```
create my-mac-address 8 c, 0 c, 20 c, 0 c, 14 c, 5e c,
my-mac-address 6 encode-bytes " local-mac-address"  property
```

In many systems, the MAC address is not associated with the individual network devices, but instead with the system itself. In such cases, the system-wide MAC address applies to all the network interfaces on that system, and individual network device nodes might not have `local-mac-address` properties. In other cases, especially with plug-in network interface cards that are intended for use on a variety of different systems, the manufacturer of the card assigns a MAC address to the card, which is reported via the `"local-mac-address"` property. A system is not obligated to use that assigned MAC address if it has a system-wide MAC address.

See also: `"network"`, `"mac-address"`, `mac-address`

loop

stack: (C: dodest-sys --)
(--) (R: loop-sys1 -- <nothing> | loop-sys2)
code: none
generates: `b(loop) -offset`

Terminates a `do` or `?do` loop. Increments the loop index by one. If the index was incremented across the boundary between *limit-1* and *limit*, the loop is terminated and loop control parameters are discarded. When the loop is not terminated, execution continues just after the corresponding `do` or `?do`.

For example, the following `do` loop:

```
8 0 do…loop
```

terminates when the loop index changes from 7 to 8. Thus, the loop will iterate with loop index values from 0 to 7, inclusive.

`loop` may be used either inside or outside of colon definitions.

+loop

stack: (C: dodest-sys --)
(n --) (R: loop-sys1 -- <nothing> | loop-sys2)
code: none
generates: `b(+loop) -offset`

Terminates a `do` or `?do` loop. Increments the loop index by *n* (or decrements the index if *n* is negative). If the index was incremented (or decremented) across the boundary between *limit-1* and *limit* the loop is terminated and loop control parameters are discarded. When the loop is not terminated, execution continues just after the corresponding `do` or `?do`.

The following `do` loop:

```
8 0 do…2 +loop
```

terminates when the loop index crosses the boundary between 7 and 8. Thus, the loop will iterate with loop index values of 0, 2, 4, 6.

By contrast, a `do` loop created as follows:

```
0 8 do…-2 +loop
```

terminates when the loop index crosses the boundary between -1 and 0. Thus, the loop will iterate with loop index values of 8, 6, 4, 2, 0.

`+loop` may be used either inside or outside of colon definitions.

lpeek

stack: (qaddr -- false | quad true)
code: 222

Tries to read the 32-bit word at address *qaddr*. Returns *quad* and *true* if the access was successful. A *false* return indicates that a read access error occurred. *qaddr* must be 32-bit aligned.

lpoke

stack: (quad qaddr -- okay?)
code: 225

Tries to write *quad* at address *qaddr*. Returns *true* if the access was successful. A *false* return indicates a read access error. *qaddr* must be 32-bit aligned.

Note – `lpoke` may be unreliable on bus adapters that “buffer” write accesses.

ls

stack: (--)
code: none

Displays the names of the active package's children.

lshift

stack: (x1 u -- x2)
code: 27

Shifts *x1* left by *u* bit-places. Zero-fills the low bits.

lwflip

stack: (quad1 -- quad2)
code: 226

Swaps the doublets within a quadlet.

lwflips

stack: (qaddr len --)
code: 237

Swaps the order of the 16-bit words within each 32-bit word in the memory buffer *qaddr len*. *qaddr* must be four-byte-aligned. *len* must be a multiple of `/l`.

For example:

```
ok h# 12345678 8000 l!
ok 8000 4 lflips
ok 8000 l@ .h
56781234
```

lwsplit

stack: (quad -- w1.lo w2.hi)
code: 7C

Splits the 32-bit value *quad* into two 16-bit words. All but the least significant 16 bits of each stack result are zero.

lxjoin

stack: (quad.lo quad.hi -- o)
code: 243

Join 2 quadlets to form an octlet.The high-order bits of each of the quadlets are ignored.

This function is only available on 64-bit implementations.

m*

stack: (n1 n2 -- d.prod)
code: none

Performs a signed multiply with a double-number product.

mac-address

stack: (-- mac-str mac-len)
code: 1A4

Usually used only by the `"network"` device type, this FCode returns the value for the *Media Access Control*, or MAC address, that this device should use for its own address. The data is encoded as a byte array, generally 6 bytes long.

The value returned by `mac-address` is system-dependent.

See also: `"mac-address"`, `"local-mac-address"`, and `"network"` in Chapter 5 “Properties” and Chapter 8 “Network Devices”.

"mac-address"

This property specifies the 48-bit IEEE 802.3-style MAC address that was last used by the device represented by the package, of device type `"network"`, containing this property. This property is created by the `open` method of a `"network"` device.

The property value is an array of six bytes encoded with `encode-bytes`.

This property is typically used by client programs that need to determine which network address was used by the network interface from which the client program was loaded.

make-properties

stack: (--)
code: none

This User Interface word is intended to be used for debugging FCode within the context of `begin-package`...`end-package`. Executing this word creates the default PCI bus properties for the current instance from information contained in the PCI Configuration Space header. This word should be executed before evaluating the FCode for the node.

map

stack: (phys.lo … phys.hi virt len mode … --)
code: none

Creates an address translation associating virtual addresses beginning at *virt* and continuing for *len* bytes with consecutive physical addresses beginning at phys.lo … phys.hi. The physical address format is the same as that of the `/memory` node. *mode* … is an MMU-dependent parameter (typically, but not necessarily, one cell) denoting additional attributes of the translation, for example access permissions, cacheability, mapping granularity, etc. If all *mode* cells have the value -1, an MMU dependent default mode is used. If there are already existing address translations within the region delimited by *virt* and *len*, the result is undefined.

If the operation fails for any reason, `map` uses `throw` to signal the error.

See also: `claim`, `modify`, `release`, `translate`

map-in

stack: (phys.lo … phys.hi size -- virt)
code: none

Creates a mapping associating the range of physical addresses beginning at *phys.lo* … *phys.hi* and extending for *size* bytes within this device's physical address space with a processor virtual address. Returns that virtual address *virt*.

The number of cells in the list *phys.lo* … *phys.hi* is determined by the value of the `"#address-cells"` property of the node containing `map-in`.

For example, to map the registers of a PCI device with:

- A register field at 10.0000-10.00ff in memory space that is controlled by the first 32-bit base address register.
- A register field at 20.0000-20.037f in I/O space that is controlled by the second 32-bit base address register.
- A non-relocatable field at 0-fff in I/O space.

use the following:

```
my-address 10.0000 0 d+ my-space 0200.0010 or 100
" map-in" $call-parent to mem-virt
my-address 20.0000 0 d+ my-space 0100.0014 or 380
" map-in" $call-parent to io-virt
my-address my-space h# 8100.0000 or 1000
" map-in" $call-parent to non-reloc-virt
```

Note – Although the third register field is non-relocatable, it is still necessary to map the address range to obtain a virtual address.

Note – It is not necessary to map the configuration registers since they can be directly addressed by using `my-space` and the `config-`*xx* family of methods.

If `map-in` cannot perform the requested operation, `throw` is called with an appropriate error message. Therefore, out-of-memory conditions can be detected and handled properly with the phrase: `['] map-in catch`

See also: `config-l@`, `map-low`, `map-out`, `my-space`

map-low

stack: (phys.lo … size -- virt)
code: 130

Creates a mapping associating the range of physical addresses beginning at *phys*.lo … *my*-space and extending for *size* bytes within this device's physical address space with a processor virtual address. Return that virtual address *virt*.

Equivalent to:

```
my-space swap " map-in" $call-parent
```

The number of cells in the list *phys.lo* … is one less than the number determined by the value of the "`#address-cells`" property of the parent node.

If the requested operation cannot be performed, `throw` is called with an appropriate error message.

Out-of-memory conditions can be detected and handled with the phrase:
`[']` `map-low` `catch`

See also: `map-out`

map-out

stack: (virt size --)
code: none

Destroys the mapping set up by a previous `map-in` at virtual address *virt*, of length *size* bytes.

See also: `free-virtual`, `map-in`

mask

stack: (-- a-addr)
code: 124

This `variable` defines which bits out of every 32-bit word will be tested by `memory-test-suite`. To test all 32-bits, set `mask` to all ones with:

```
ffff.ffff mask !
```

To test only the low-order byte out of each word, set the lower bits of `mask` with:

```
0000.00ff mask !
```

Any arbitrary combination of bits can be tested or masked.

max

stack: (n1 n2 -- n1|n2)
code: 2F

Returns the greater of *n1* and *n2*.

"max-frame-size"

This property, when declared in "`network`" devices, indicates the maximum packet length (in bytes) that the physical layer of the device can transmit at one time. This value can be used by client programs to allocate buffers of the appropriate length.

Used as:

```
4000 encode-int " max-frame-size" property
```

max-transfer

stack: (-- max-len)
code: none

Returns the size in bytes of the largest single transfer that this device can perform, rounded down to a multiple of `block-size`.

"memory"

This is the standard property value of the `"device_type"` property for memory devices. Devices of type `"memory"` must implement the following methods:

- `claim`
- `release`

See *IEEE Standard 1275-1994* for more details.

See also: `alloc-mem`, `"available"`, `claim`, `"reg"`, `release`, `"#size-cells"`

memory-test-suite

stack: (addr len -- fail?)
code: 122

Performs a series of tests on the memory beginning at *addr* for *len* bytes. If any of the tests fail, `failed?` is `true` and a failure message is displayed on a system-dependent diagnostic output device.

The actual tests performed are machine specific and often vary depending on whether `diagnostic-mode?` is `true` or `false`. Typically, if `diagnostic-mode?` is `true`, a message is sent to the console output device giving the name of each test.

The value stored in `mask` controls whether only some or all data lines are tested.

For example:

```
: test-result ( -- )
   frame-buffer-adr my-frame-size memory-test-suite  ( failed? )
    encode-int “ test-result” property
;
```

See also: `diag-switch?`

min

stack: (n1 n2 -- n1|n2)
code: 2E

Returns the lesser of *n1* and *n2*.

mod

stack: (n1 n2 -- rem)
code: 22

rem is the remainder after dividing *n1* by the divisor *n2*. `rem` has the same sign as *n2* or is zero. An error condition results if the divisor is zero.

*/mod “star slash mod”

stack: (n1 n2 n3 -- rem quot)
code: none

Calculates *n1* `*` *n2* `/` *n3* and returns the remainder and quotient. The inputs, outputs, and intermediate products are all 32-bit. *rem* has the same sign as *n3* or is zero. An error condition results if the divisor is zero.

/mod “slash mod”

stack: (n1 n2 -- rem quot)
code: 2A

rem is the remainder and *quot* is the quotient of *n1* divided by the divisor *n2*. *rem* has the same sign as *n2* or is zero. An error condition results if the divisor is zero.

model

stack: (str len --)
code: 119

This is a shorthand word for creating a `"model"` property. By convention, `"model"` identifies the model name/number for a PCI card, for manufacturing and field-service purposes. A sample usage would be:

```
" INTL,501-1415-1" model
```

This is equivalent to:

```
" INTL,501-1415-1" encode-string  " model" property
```

The `"model"` property is useful to identify the specific piece of hardware (the PCI card), as opposed to the `"name"` property (since several different but functionally-equivalent cards would have the same `"name"` property, thus calling the same operating system device driver).

See also: `property`, `"model"` in Chapter 5 “Properties”.

"model"

This property specifies the model name and number (including revision level) for this device in a manufacturer-dependent string. The format of the text string is arbitrary, although in conventional usage the string begins with the name of the device’s manufacturer as with the `"name"` property.

Although there is no standard interpretation for the value of the `"model"` property, a specific device driver might use it to learn, for instance, the revision level of its particular device.

For example:

```
" AAPL,1416-02" encode-string  " model" property
```

See also: `property`, `model`

modify

stack: (virt len mode … --)
code: none

Modifies the existing address translations for virtual addresses beginning at *virt* and continuing for *len* bytes to have the attributes specified by *mode* … (whose format depends upon the package).

If the operation fails for any reason, uses `throw` to signal the error.

See also: `claim`, `map`, `release`, `translate`, `unmap`

move

stack: (src_addr dest_addr len --)
code: 78

len bytes starting at *src_addr* (through *src_addr+len-1* inclusive) are moved to address *dest_addr* (through *dest_addr+len-1* inclusive). If *len* is zero then nothing is moved.

The data are moved such that the *len* bytes left starting at address *dest_addr* are the same data as was originally starting at address *src_addr*. If *src_addr* > *dest_addr* then the first byte of *src_addr* is moved first, otherwise the last byte (*src_addr+len-1*) is moved first. Thus, moves between overlapping fields are properly handled.

`move` will perform 16-bit, 32-bit or possibly even 64-bit operations (for better performance) if the alignment of the operands permits. If your hardware requires explicit 8-bit or 16-bit accesses, you will probably wish to use an explicitly-coded `do` … `loop` instead.

ms

stack: (n --)
code: 126

Delays all execution for at least *n* milliseconds, by executing an empty delay loop for an appropriate number of iterations. The maximum allowable delay will vary from system to system, but is guaranteed to be valid for all values up to at least 1,000,000 (decimal). No other CPU activity takes place during delays invoked with `ms`, although generally this is not a problem for FCode drivers since there is nothing else to do in the meantime anyway. If this word is used excessively, noticeable delays could result.

For example:

```
: probe-loop-wait ( addr -- )
   \ wait h# 10 ms before doing another probe at the location
   begin  dup l@ drop h# 10 ms key?  until  drop
;
```

my-address

stack: (-- phys.lo …)
code: 102

Returns the low component(s) of the device’s probe address, suitable for use with the `map-in` method, and with `reg` and `encode-phys`. The returned number, along with `my-space`, encodes the address of location 0 of this device in a bus-specific format. The number of cells in the list *phys.lo* … is one less than the number determined by the value of the `"#address-cells"` property of the parent node.

The Open Firmware ROM automatically sets `my-address` to the correct value before each slot is probed. Usually, this value is used to calculate the location(s) of the device registers, which are then saved as the property value of the `"reg"` property and later accessed with `my-unit`.

For example for a PCI device:

```
fcode-version2
  " audio" encode-string " name" property
  my-address my-space encode-phys           \ PCI Configuration Space
  0 encode-int encode+  0 encode-int encode+
  …
  " reg" property
end0
```

my-args

stack: (-- arg-str arg-len)
code: 202

Returns the instance argument string *arg-str arg-len* that was passed to the current instance when it was created, if the argument string exists. Otherwise returns with a length of 0.

For example:

```
ok " /obio:TEST-ARGS" open-dev to my-self my-args type
TEST-ARGS
ok unselect-dev " /obio:MORE-ARGS" select-dev my-args type
MORE-ARGS
```

my-parent

stack: (-- ihandle)
code: 20A

Returns the *ihandle* of the instance that opened the current instance. For device driver packages, the relationships of parent/child instances mimic the parent/child relationships in the device tree.

For example for an SBus device:

```
: show-parent ( -- )
   my-parent ihandle>phandle " name" rot
   get-package-property 0= if
     ." my-parent is " type cr
   then
;
```

my-self

stack: (-- ihandle)
code: 203

A `value` word that returns the current instance's *ihandle*. If there is no current instance, the value returned is zero.

For example:

```
: show-model-name ( -- )
   my-self ihandle>phandle ( phandle )
   " model" rot get-package-property 0= if  ( val.addr,len )
      ." model name is " type cr
   else  ( )
      ." model  property is missing " cr
   then ( )
;
```

my-space

stack: (-- phys.hi)
code: 103

Returns the high component of the device's probe address representing the device space that this card is plugged into. The meaning of the returned value is bus-specific.

For example for an SBus device:

```
fcode-version2
  " audio" encode-string " name" property
  my-address h# 130.0000 + my-space h# 8 reg
  …
fcode-end
```

See `my-address` for more details.

my-unit

stack: (-- phys.lo … phys.hi)
code: 20D

Returns the unit address *phys.lo* … *phys.hi* of the current instance. The unit address is set when the instance is created, as follows:

- If the *node-name* used to locate the instance's package contained an explicit *unit-address*, that is the instance's unit address. This handles the case of a "wildcard" node with no associated `"reg"` property.
- Otherwise, if the device node associated with the package from which the instance was created contains a `"reg"` property, the first component of its `"reg"` property value is the instance's unit address.
- Otherwise, the instance's unit address is 0 0.

The number of cells in the list `phys.lo` … `phys.hi` is determined by the value of the `"#address-cells"` property of the parent node.

/n "per en"

stack: (-- n)
code: 5D

The number of address units in a cell.

/n* “per en star”

stack: (nu1 -- nu2)
generates: `cells`

Synonym for `cells`.

na+ “en ay plus”

stack: (addr1 index -- addr2)
code: 61

Increments *addr1* by *index* times the value of `/n`.

`na+` should be used in preference to `wa+` or `la+` when the intent is to address items that are the same size as items on the stack.

na1+ “en ay one plus”

stack: (addr1 -- addr2)
generates: `cell+`

Synonym for `cell+`.

"name"

This property specifies the manufacturer’s name and device name of the device. All device nodes *must* publish this property. The `"name"` property can be used to match a particular operating system device driver with the device.

The property value is an arbitrary string. Any combination of one to 31 printable characters is allowed, except for “@”, “:” or “/”. The string may contain at most one comma. Embedded spaces are not allowed.

IEEE Standard 1275-1994 specifies three different formats for the manufacturer’s name portion of the property value where two of those formats are strongly preferred.

For United States companies that have publicly listed stock, the most practical form of name is to use the company’s stock symbol (e.g. AAPL for Apple Computer, Inc.). This option is also available to any company anywhere in the world provided that there is no duplication of the company’s stock symbol on either the New York Stock Exchange or the NASDAQ exchange. If a non-U.S. company’s stock is traded as an American Depository Receipt (ADR), the ADR symbol satisfies this requirement. A prime advantage of this form of manufacturer’s name is that such stock symbols are very human-readable.

An alternative is to obtain an *organizationally unique identifier* (OUI) from the IEEE Registration Authority Committee. This is a 24-bit number that is guaranteed to be unique world-wide. Companies that have obtained an OUI would use a sequence of hexadecimal digits of the form “0NNNNNN” for the manufacturer’s name portion of the property. This form of name has the disadvantage that the manufacturer is not easily recognizable.

Each manufacturer may devise its own format for the device name portion of the property value.

An example usage is:

```
" INTL,bison-printer" encode-string " name" property
```

The `device-name` command may also be used to create this property.

See also: `device-name`, `property`, `"name"` in Chapter 5 “Properties”.

named-token

stack: (--) (F: /FCode-string FCode#/ --)
code: B6

Creates a new, possibly-named FCode function. `named-token` should never be used directly in source code.

negate

stack: (n1 -- n2)
code: 2C

n2 is the negation of *n1*. This is equivalent to `0 swap -` .

"network"

This is the standard property value of the `"device_type"` property for network devices with IEEE 802 packet formats.

Devices of type `"network"` must implement the following methods:

- `open`
- `close`
- read

  The `read` method receives (non-blocking) a network packet placing at most the first *len* bytes into memory at *addr*, returning either the number of bytes actually received (not placed into memory) or -2 if no packet is currently available.

Note – In general, -2 indicates no data was available at the time `read` was done and -1 indicates that an error occurred. Zero is generally used only for devices where data arrives in records, packets or other such container, and indicates that a valid but empty container was received.

- write

  The `write` method transmits the network packet of *len* bytes from memory at *addr*, returning the number of bytes actually transmitted. The caller must supply the complete packet including the MAC header with source and destination address.

- load

A `network` package may implement additional device-specific methods.

See also: `"address-bits"`, `"max-frame-size"`

new-device

stack: (--)
code: 11F

Creates a new node in the device tree as a child of the active package and makes the new node the active package. Also creates a new instance and attaches that instance to the instance currently identified by `my-self` (i.e. the new node’s parent node).

Subsequently, newly-defined Forth words become the methods of the node created by `new-device` and newly-defined data items (such as types `variable`, `value`, `buffer:` and `defer`) are allocated and stored with the new instance.

`new-device` is used for creating multiple devices in a single FCode Program.

See also: `finish-device`, `begin-package`

new-token

stack: (--) (F: /FCode#/ --)
code: B5

Creates a new unnamed FCode function. `new-token` should never be used directly in source code.

next-property

stack: (previous-str previous-len phandle -- false | name-str name-len true)
code: 23D

Returns the `name` of the property following *previous-string* of *phandle*.

Locates with the property list of the package specified by *phandle*, the first property if *previous-len* is zero, or the property following the property specified by *previous-string* otherwise. If such a property exists, *name-string* is returned underneath `true`. Otherwise, `false` is returned (i.e. if there are no more properties, or if *previous-string* specifies a property which does not exist in *phandle*).

A sequence of invocations of `next-property` with the same *phandle* value, beginning with *previous-len* equal to zero, and passing the *name-string* result of the previous invocation as the *previous-string* argument to the next invocation, continuing until `false` is returned will provide the complete list of properties of the package *phandle*.

However, the order in which the properties are returned is undefined (e.g. the first property defined is not necessarily the first property returned). Consequently, if a new property is defined in the *phandle* package in the middle of the process of extracting all of the properties of the package *phandle*, the newly defined property may or may not be returned.

nip

stack: (x1 x2 -- x2)
code: 4D

Removes the second item on the stack.

nodefault-bytes

stack: (maxlen "new-name< >" --) (E: -- addr len)
code: none

Creates a custom configuration variable of size *maxlen*. nodefault-bytes creates a configuration variable whose data is of type byte-array. As with other built-in byte-array configuration variables, these user-created configuration variables can be set with `setenv` (restricted to printable characters) or `$setenv` and can be displayed with `printenv`. However, `set-default` and `set-defaults` have no effect on user-created configuration variables.

Although the values of user-created configuration variables persist across system resets, Open Firmware must be “reminded” of their existence after every system reset in order for them to be accessed. Furthermore, the `nodefault-bytes` commands creating them must be executed in the same order each time. For these reasons, `nodefault-bytes` is usually executed from the NVRAM script.

If `nodefault-bytes` fails, `throw` is called with an appropriate error message. Consequently, out-of-memory conditions may be detected and handled properly with the phrase: `['] nodefault-bytes catch`

For example:

```
ok 100 nodefault-bytes new-name
ok setenv new-name " foo"
new-name = 22 20 60 6f 6f
ok printenv new-name
new-name = 22 20 60 6f 6f
```

noop

stack: (--)
code: 7B

Does nothing. This can be used to provide short delays or as a placeholder for patching in other commands later.

noshowstack

stack: (… -- …)
code: none

Turns off `showstack` (i.e. automatic stack display).

The system default is `noshowstack`.

See also: `showstack`

not

stack: (x1 -- x2)
generates: `invert`

Synonym for `invert`.

See also: `0=`

not-last-image

stack: (--)
generates: `nothing`

A FirmWorks extension to the tokenizer. Executing `not-last-image` prior to executing `pci-header` causes the PCI header’s “indicator” field to be set to 0 indicating the presence of a following image in the PCI Expansion ROM.

See also: `pci-header`, `pci-header-end`

$number

stack: (addr len -- true | n false)
code: A2

A numeric conversion primitive that converts a string to a number, according to the current `base` value. An error flag is returned if an inconvertible character is encountered.

For example:

```
ok hex
ok " 123f" $number .s
123f 0
ok " 123n" $number .s
ffffffff
```

>number

stack: (d1 str1 len1 -- d2 str2 len2)
code: none

Converts *str1 len1* into a number on a digit-by-digit basis according to the value in `base`. As each digit is converted, *d1* is multiplied by the value of `base` and the newly-converted digit is added to *d1*.

See also: `$number`

nvalias

stack: ("alias-name< >device-specifier<eol>" --)
code: none

Creates the following command line in the script:

`devalias` *alias-name device-specifier*

If the script already contains a `devalias` line with the same alias name, that entry is deleted and replaced with the new entry at the same location in the script. Otherwise, the new entry is placed at the beginning of the script.

If there is insufficient space in the script for the new devalias command, a message is displayed to that effect and the operation is aborted without modifying the script.

If the script was successfully modified, the new `devalias` command is executed immediately, creating a new memory-resident alias.

If the script is currently being edited (i.e. `nvedit` has been executed, but has not been completed with either `nvstore` or `nvquit`), the operation is aborted with an error message before taking any other action.

If the script was successfully modified, but `use-nvramrc?` is `false`, `use-nvramrc?` is set to `true`.

For example:

```
ok nvalias alias-name /full/pathname
```

$nvalias

stack: (name-str name-len dev-str dev-len --)
code: none

Performs the same function as `nvalias`, except that parameters are stack strings. The alias name is specified by *name-string*. The device-specifier is specified by *dev-string*.

For example:

```
ok " new-alias" " device-specifier" $nvalias
```

nvedit

stack: (--)
code: none

`nvedit` operates on a temporary buffer. If data remains in the temporary buffer from a previous `nvedit`, editing will resume with those previous contents. If not, `nvedit` will read the contents of the script into the temporary buffer and begin editing the temporary buffer.

Editing continues until Control-C is typed, at which moment editing ceases and normal operation of the command interpreter is resumed. The contents of the temporary buffer are not automatically saved to the script; the `nvstore` command must be executed afterwards to save the buffer into the script.

The following table lists the command keystrokes used to edit the NVRAM script.

Table 37 NVRAM Script Editor Keystroke Commands

Keystroke	Description
Control-B	Moves backward one character.
Escape B	Moves backward one word.
Control-F	Moves forward one character.
Escape F	Moves forward one word.
Control-A	Moves backward to beginning of line.
Control-E	Moves forward to end of line.
Control-N	Moves to the next line of the script editing buffer.
Control-P	Moves to the previous line of the script editing buffer.
Return (Enter)	Inserts a newline at the cursor position and advances to the next line.
Control-O	Inserts a newline at the cursor position and stays on the current line.
Control-K	Erases from cursor to end of line, storing erased characters in a save buffer. If at the end of a line, joins the next line to the current line (i.e. deletes the newline).
Delete	Erases previous character.
Backspace	Erases previous character.
Control-H	Erases previous character.
Escape H	Erases from beginning of word to just before the cursor, storing erased characters in a save buffer.
Control-W	Erases from beginning of word to just before the cursor, storing erased characters in a save buffer.
Control-D	Erases next character.

Table 37 NVRAM Script Editor Keystroke Commands *(Continued)*

Keystroke	Description
Escape D	Erases from cursor to end of the word, storing erased characters in a save buffer.
Control-U	Erases entire line, storing erased characters in a save buffer.
Control-Y	Inserts the contents of the save buffer before the cursor.
Control-Q	Quotes next character (allows you to insert control characters).
Control-R	Retypes the line.
Control-L	Displays the entire contents of the editing buffer.
Control-C	Exits the script editor, returning to the Open Firmware command interpreter. The temporary buffer is preserved, but is not written back to the script. (Use `nvstore` afterwards to write it back.)

nvquit

stack: (--)
code: none

Prompts for confirmation of the user's intent to carry out this function. If confirmation is obtained, discards the `nvedit` temporary buffer. Otherwise, takes no further action.

nvramrc

stack: (-- data-addr data-len)
code: none

Returns the location and size of the NVRAM script. The size of the script region is system dependent.

While it is possible to alter the contents of the script with `setenv` or `$setenv`, use of the script editor, `nvedit`, is preferred.

The contents of the script are cleared by `set-defaults`. Under some circumstances cleared contents can be recovered with `nvrecover`.

The commands in the script are interpreted during system start-up only if `use-nvramrc?` is `true`.

See also: `nodefault-bytes`, `nvedit`, `use-nvramrc?`

nvrecover

stack: (--)
code: none

Attempts to recover the contents of the script if they have been lost as a result of the execution of `set-default` or `set-defaults`. Enters the script editor as with the `nvedit` command.

In order for `nvrecover` to succeed, `nvedit` must not have been executed between the time that the script contents were lost and the time that `nvrecover` is executed.

nvrun

stack: (--)
code: none

Executes the contents of the `nvedit` temporary buffer.

nvstore

stack: (--)
code: none

Copies the contents of the `nvedit` temporary buffer into the script. The `nvedit` temporary buffer is then cleared. Use after `nvedit` to save the results of an editing session into the script.

nvunalias

stack: ("alias-name< >" --)
code: none

Delete non-volatile device alias from the script.

If the script contains a `devalias` command line with the same name as *alias-name*, deletes that command line from the script. Otherwise, leaves the script unchanged. If the script is currently being edited (i.e. `nvedit` has been executed, but has not been completed with either `nvstore` or `nvquit`), aborts with an error message before taking any other action.

For example:

```
ok nvunalias alias-name
```

$nvunalias

stack: (name-str name-len --)
code: none

Deletes the specified non-volatile device alias from the script. Similar to `nvunalias` except that the alias name is specified by *name-string*.

For example:

```
ok " alias-name" $nvunalias
```

o# “oh number”

stack: ([number< >] -- n)
generates: `b(lit)` `xx-byte` `xx-byte` `xx-byte` `xx-byte`

Interprets the next number in octal (base 8), regardless of any previous settings of `hex`, `decimal` or `octal`. Only the immediately following number is affected; the default numeric base setting is unchanged. For example:

```
hex
o# 100 ( equals decimal 64 )
o# 100 ( equals decimal 64 )
100 ( equals decimal 256 )
```

See also: `d#`, `h#`

"obp-tftp"

This standard package implements the Internet Trivial File Transfer Protocol (TFTP) for use in network booting. The package is typically used by `network` device drivers.

The `obp-tftp` package uses the `read` and `write` methods of the package that opened it and implements the following methods:

- `open ( -- okay? )`

  Prepare this device for subsequent use.

- `close ( -- )`

  Close this previously-`open`'d device.

- `load ( addr -- size )`

  Load a client program from device to memory.

octal

stack: (--)
code: none

If `octal` is encountered by the tokenizer in FCode Source outside a definition, the tokenizer sets its numeric conversion radix to eight.

If `octal` is encountered by the tokenizer in FCode Source inside a definition, the tokenizer appends the following sequence to the FCode Program that is being created: `8 base !` This affects numeric output when the FCode Program is later executed.

See also: `base`

oem-banner

stack: (-- text-str text-len)
code: none

The value of this configuration variable is a string containing the custom banner text, the display of which is controlled by the configuration variable `oem-banner?`.

The suggested default value is an empty string.

oem-banner?

stack: (-- custom?)
code: none

This configuration variable is a boolean specifying whether to display a custom message instead of the normal system-dependent messages. If `oem-banner?` is `true`, `banner` displays the value of `oem-banner`. If `oem-banner?` is `false`, `banner` displays the normal system-dependent messages.

The suggested default value of `oem-banner?` is `false`.

oem-logo

stack: (-- logo-addr logo-len)
code: none

This configuration variable contains a 512 byte array which holds a bit map of a custom logo. The custom logo is displayed if the configuration variable `oem-logo?` is `true`.

The logo is a 512-byte field, representing a 64x64-bit logo bit map. Each bit controls one pixel. The most significant bit of the first byte controls the upper-left corner pixel. The next bit controls the next pixel to the right and so on.

`oem-logo` is unaffected by `set-default` or `set-defaults`.

`oem-logo` cannot receive arbitrary data with `setenv`, but `$setenv` can be used to set its value. For example:

```
( logo-addr logo-len ) " oem-logo" $setenv
```

The suggested default value is all zeroes.

oem-logo?

stack: (-- custom-logo?)
code: none

This configuration variable is a boolean specifying whether to display a custom logo instead of the normal system-dependent logo. If `oem-logo?` is `true`, `banner` displays the value of `oem-logo`. If `oem-logo?` is `false`, `banner` displays the normal system-dependent logo.

The suggested default value of `oem-logo?` is `false`.

of

stack: (C: case-sys1 -- case-sys2 of-sys)
(sel of-val -- sel | <nothing>)
generates: `b(of) +offset`

Begins the next test clause in a `case` statement. See `case` for more details.

off

stack: (a-addr --)
code: 6B

Sets the 32-bit contents at *a-addr* to *false* (i.e. zero).

offset

stack: (d.rel -- d.abs)
code: none

This method of the Disk Label Support Package converts a partition-relative disk position to an absolute position. *d.rel* is a double-number disk position, expressed as the number of bytes from the beginning of the partition that was specified in the arguments when the support package was opened. *d.abs* is the corresponding double-number disk position, expressed as the number of bytes from the beginning of the disk. If no partition was specified when the support package was opened, a system-dependent default partition is used.

If the Disk Label Support Package does not support disk partitioning, *d.abs* is equal to *d.rel*.

offset16

stack: (--)
code: CC

Instructs the tokenizer program, and the boot ROM, to expect all further branch offsets to be 16-bit values. This word is automatically generated by some current tokenizers.

Once `offset16` is executed, the offset size remains 16 bits for the duration of the FCode Program; it cannot be set back to 8 bits. Multiple calls of `offset16` have no additional effect. `offset16` is only useful within an FCode Program that begins with `version1`. All other starting tokens (`start0`, `start1`, `start2`, and `start4`) automatically set the offset size to 16 bits.

See also: `fcode-version2`

on

stack: (a-addr --)
code: 6A

Set the 32-bit contents at *a-addr* to *true* (i.e. `ffffffff`).

open

stack: (-- okay?)
code: none

Prepares this device for subsequent use. Typical behavior is to allocate any special resource requirements it needs, map the device into virtual address space, initialize the device and perform a brief "sanity test" to ensure that the device appears to be working correctly.

Returns *true* if `open` was successful, *false* if not.

When a device's `open` method is called, that device's parent has already been opened (and so on, up to the root node, which has no parent), so this `open` method can call its parent's methods, for instance to create mappings within the parent's address space.

open-dev

stack: (dev-str dev-len -- ihandle | 0)
code: none

Opens the device specified by *dev-string*. Returns *ihandle* if successful, or 0 if not. Opens each node of the device tree in turn, starting at the top. The current instance and the active package are not changed.

For example:

```
" device-alias" open-dev
```

See also: `my-self`

open-package

stack: (arg-str arg-len phandle -- ihandle | 0)
code: 205

Creates an instance of the package identified by *phandle*, saves in that instance an argument string specified by *arg-str arg-len*, and invokes the package's `open` method. The parent instance of the new instance is the instance that invoked `open-package`.

Returns the instance handle *ihandle* of the new instance if it can be opened. It returns 0 if the package could not be opened, either because that package has no `open` method or because its `open` method returned *false* indicating an error. In this case, the current instance is not changed.

For example:

```
: test-tftp-open ( -- ok? )
   " obp-tftp" find-package  if  ( phandle )
      0 0 rot open-package  if  true  else  false  then
   else
      false
   then
;
```

See also: `close-package`

$open-package

stack: (arg-str arg-len name-str name-len -- ihandle | 0)
code: 20F

Similar to using `find-package` `open-package` except that if `find-package` fails, 0 is returned immediately, without calling `open-package`.

The name is interpreted relative to the `/packages` device node. For example, if *name-str name-len* represents the string `"disk-label"`, the package in the device tree at "`/packages/disk-label`" will be located.

If there are multiple packages with the same name (within the `/packages` node), the most recently created one is opened.

For example:

```
0 0  " obp-tftp" $open-package  ( ihandle )
```

See also: `close-package`

"/openprom"

The standard node describing the system's Open Firmware. The value of the `"name"` property of this node is "`openprom`". The remaining standard properties of this node are:

- model
- relative-addressing

Other system-dependent properties may also be present.

"/options"

The standard node containing the system's configuration variables. The value of the `"name"` property of this node is "`options`".

The names of the remaining properties of this node are the names of the configuration variables. The values of the remaining properties of this node are the current settings of the configuration variables.

Client programs may examine and change the values of these properties with the Client Interface’s `getprop`, `nextprop` and `setprop` services, thus examining and changing the values of the corresponding configuration variables. Similarly, users may examine and change them with `printenv`, `setenv`, and `$setenv`.

or

stack: (x1 x2 -- x3)
code: 24

x3 is the bit-by-bit inclusive-or of *x1* with *x2*.

#out

stack: (-- a-addr)
code: 93

A `variable` containing the current column number on the output device. This is updated by `emit`, `cr` and some other words that modify the cursor position. It is used for display formatting.

For example:

```
: to-column  ( column -- )  #out @  -  1 max spaces ;
```

output

stack: (dev-str dev-len --)
code: none

Selects the specified device for console output as follows:

- Searches for a device node matching the pathname or device-specifier given by *dev-str dev-len*.
- If such a device is found, search for its `write` method.
- If the `write` method is found, open the device with `open-dev`.
- If the open succeeds, execute the device’s `install-abort` method, if any.
- If any of these steps fails, display an appropriate error message and return without performing the steps following the one that failed.

If there is a console output device, as indicated by a non-zero value in the `stdout` variable, `output` executes the current console output device’s `remove-abort` method and closes the console output device. `output` then sets `stdout` to the ihandle of the newly opened device, making it the new console output device.

For example:

```
" device-alias" output
```

output-device

stack: (-- dev-str dev-len)
code: none

The value of this configuration variable is a string specifying the console output device to be established by install-console. *dev-string* is a device-specifier, containing either a full device-path or a pre-defined device alias.

The suggested default value is “screen”.

For example:

```
ok setenv output-device device-alias
```

over

stack: (x1 x2 -- x1 x2 x1)
code: 48

The second stack item is copied to the top of the stack.

2over

stack: (x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2)
code: 54

Copies the third and fourth stack items to the stack top.

pack

stack: (str len addr -- pstr)
code: 83

Stores the string specified by *str len* as a packed string at the location *addr* returning *pstr* (which is the same address as *addr*). The byte at address *pstr* is the length of the string and the string itself starts at address *pstr+1*. A packed string can contain at most 255 characters.

Packed strings are generally not used in FCode. Virtually all string operations are in the *addr len* format.

For example:

```
h# 20 buffer: my-packed-string
" This is test string " my-packed-string pack
```

"/packages"

The standard node containing the system’s support packages (both standard and system-specific). The value of the `"name"` property of this node is “`packages`”.

The children of this node are general-purpose support packages not attached to any particular hardware device. The “physical address space” defined by this node is the trivial one i.e. all addresses are the same (0,0). Its children are distinguished by name alone.

For example, the `disk-label` support package is located in the device tree at `/packages/disk-label`.

"page-size"

This `/mmu` property defines the virtual address space page size.

The property value is an integer specifying the number of bytes in the smallest mappable region of virtual address space. For example:

```
d# 4096 encode-int " page-size" property
```

parse

stack: (delim "text<delim>" -- str len)
code: none

Parses text from the input buffer, delimited by *delim*. For example:

```
: dir"  ( "pattern" -- )  [char] " parse $dir  ;
```

parse-2int

stack: (str len -- val.lo val.hi)
code: 11B

Converts a “*hi,lo*” string into a pair of values according to the current value in `base`.

If the string does not contain a comma, *val.lo* is zero and *val.hi is* the result of converting the entire string. If either component contains non-numeric characters, according to the value in `base`, the result is undefined.

For example:

```
ok " 4,ff001200" parse-2int .s
ff001200 4
ok " 4" parse-2int .s
0 4
```

parse-word

stack: ("text< >" -- str len)
code: none

Parses text from the input buffer, delimited by white space after skipping any leading spaces. *str* is the address (within the input buffer) and *len* is the length of the selected string. If the parse area was empty, the resulting string has a zero length.

password

stack: (--)
code: none

Prompts the user (twice) to enter a new password, terminated by end-of-line. Does not echo the password on the screen as it is typed. The password length is zero to eight characters in length. Ignores any additional characters (more than eight).

If the entered password is the same both times, stores the new password string in `security-password`. Note that `security-mode` must be set to enable password protection.

patch

stack: ("new-name< >old-name< >word-to-patch< >" --)
code: none

In the compiled definition of *word-to-patch*, changes the first occurrence of *old-name* to *new-name*. Works properly even if *old-name* and / or *new-name* are numbers.

For example:

```
ok : patch-me  test 0 do  i . cr  loop ;
ok patch 555 test patch-me
ok see patch-me
: patch-me
   h#555 0 do
      i . cr
   loop
;
```

See also: `(patch)`.

(patch)

stack: (new-n1 num1? old-n2 num2? xt --)
code: none

Change contents of command indicated by *xt*.

In the compiled definition of the command indicated by *xt*, changes the first occurrence of *old-n2* to *new-n1*. *new-n1* and *old-n2* can each be either an execution token or a literal number. The flag *num1?* is *true* if *new-n1* is a literal number. If *false*, it indicates that *new-n1* is an execution token. The flag *num2?* is interpreted similarly.

For example:

```
ok : patch-me  555 0 do  i . cr  loop ;
ok ['] new-name false 555 true ['] patch-me (patch)
ok see patch-me
: patch-me
   new-name 0 do
      i . cr
   loop
;
```

See also: `patch`

peer

stack: (phandle -- phandle.sibling)
code: 23C

`peer` returns the phandle *phandle.sibling* of the package that is the next child of the parent package *phandle*.

If there are no more siblings, `peer` returns 0.

If *phandle* is 0, `peer` returns *phandle* of the root node.

Together with `child`, `peer` lets you enumerate (possibly recursively) the children of a particular device. A common application would be for a device driver to use `child` to determine the phandle of a node's first child, and use `peer` multiple times to determine the phandles of the node's other children. For example:

```
: my-children ( -- )
  my-self ihandle>phandle child  ( first-child )
  begin ?dup while dup . peer repeat
;
```

pci-header

stack: (vendor-id device-id class-code --)
generates: PCI Expansion ROM header

A FirmWorks extension to the tokenizer. Executing `pci-header` results in the creation of a PCI Expansion ROM header. In addition to filling in the header's "vendor ID", "device ID" and "class code" fields with the values supplied by its stack arguments, `pci-header` puts a default value of 0 into the "pointer to vital product data" field, puts a default "1" in the "revision level" field and sets the "indicator" field to a default value of 1 indicating that this is the last image in the ROM. `pci-header` must be paired with `pci-header-end` to create a complete PCI Expansion ROM header.

The macros `set-rev-level`, `set-vpd-offset`, and `not-last-image` are provided to override the default values used by `pci-header`.

See also: `fcode-end`, `pci-header-end`, `not-last-image`, `set-rev-level`, `set-vpd-offset`

pci-header-end

stack: (--)
generates: "image length" field of PCI Expansion ROM header

A FirmWorks extension to the tokenizer. `pci-header-end` computes the correct value for the "image-length" field of the PCI Expansion ROM header by rounding up the result of dividing the length in bytes of the PCI Expansion ROM FCode image by 512. `pci-header-end` places this result in the "image length" field. `pci-header` must have been previously executed to create the PCI Expansion ROM header.

See also: `fcode-end`, `pci-header`, `not-last-image`, `set-rev-level`, `set-vpd-offset`

pick

stack: (xu … x1 x0 u -- xu … x1 x0 xu)
code: 4E

Copies *xu*, the *u+1*-th stack value, not including *u* itself, where the remaining stack items have indices beginning with 0. *u* must be between 0 and two less than the total number of elements on the stack (including *u*).

```
0 pick is equivalent to dup    ( n1 -- n1 n1 )
1 pick is equivalent to over   ( n1 n2 -- n1 n2 n1 )
2 pick is equivalent to        ( n1 n2 n3 -- n1 n2 n3 n1 )
```

For the sake of readability, the use of `pick` should be minimized.

postpone

stack: (C: [old-name< >] --)
(… -- ???)
code: none

Can be used only within definitions to delay the execution of the following word, regardless of whether or not that word is “immediate”. `postpone` affects only the behavior of the word that follows it.

```
: end-if (C: orig-sys -- ) (E: -- ) postpone then ; immediate
```

printenv

stack: ("{param-name}<eol>" --)
code: none

If *param-name* is missing, displays the current and default values of all configuration variables. Otherwise, displays the current and default values of the configuration variable given whose name is *param-name.*

probe-all

stack: (--)
code: none

Searches for plug-in devices on the system-dependent set of expansion buses, creating device nodes for devices that are located.

Undesirable results, such as duplicate device nodes for the same device, might occur if `probe-all` is executed more than once. It is normally executed automatically during system start-up following the evaluation of the script, but this automatic execution is disabled if `banner` or `suppress-banner` is executed from the script.

probe-self

stack: (arg-str arg-len reg-str reg-len fcode-str fcode-len --)
code: none

Evaluates FCode, as a child of this node.

fcode-string is a unit-address text string representing the location of the FCode Program for the child device.

reg-string is a probe-address text string representing the location of the child device itself.

arg-string is a instance-arguments text string providing the arguments for the child (which can be retrieved within the child’s FCode Program with `my-args`.)

`probe-self` first checks to see if there is an FCode Program at the indicated location (perhaps by mapping the device and using `cpeek` to ensure that the device is present and that the first byte is a valid FCode start byte). If so, `probe-self`:

- Performs the function of `new-device` (thus creating a new device node)
- Interprets the FCode Program
- Performs the function of `finish-device`

If a valid FCode Program cannot be located at the indicated address, `probe-self` does not create a new device node.

Successful completion of `probe-self` will be indicated by the presence of a new device node containing a `"name"` property. If the evaluation of the FCode Program fails in some way, the new device node might be empty (containing no properties or methods.)

.properties

stack: (--)
code: none

Displays names and values of the properties of the active package.

property

stack: (prop-addr prop-len name-str name-len --)
code: 110

Creates a new property with the specified name and previously prop-encoded value. If there is a current instance, creates the property in the package from which the current instance was created. Otherwise, if there is an active package, creates the property in the active package. If there is neither a current instance nor an active package, the result is undefined.

If a property with the specified name already exists in the active package in which the property would be created, replace its value with the new value.

Properties provide a mechanism for an FCode Program to pass information to an operating system device driver. A property consists of a property name string and a property value array. The name string gives the name of the property, and the value array gives the value associated with that name. For example, a framebuffer may wish to declare a property named `"hres"` (for horizontal resolution) with a value of 1152.

The `property` command requires two arrays on the stack — the value array and the name string. The name string is an ordinary Forth string, such as any string created with `"` . This string should be written in lower case, since the property name is stored only after converting uppercase letters, if any, to lower case. For example:

```
" A21-b" encode-string " New_verSION" property
```

is stored as if entered

```
" A21-b" encode-string " new_version" property
```

The value array, however, *must* be in the property value array format. See Chapter 5 “Properties” for more information on creating property value arrays.

All properties created by an FCode Program are stored in a “device tree” by Open Firmware. This tree can then be queried by an operating system device driver, using the Client Interface’s `getprop` or `nextprop` services.

The FCode Program and the operating system device driver may agree on any arbitrary set of names and values to be passed, with virtually no restrictions. Several property names, though, are reserved and have specific meanings. For many of them, a shorthand command also exists that makes the property declaration a bit simpler.

For example:

```
" AAPL,new-model" encode-string model
```

See also: `"name"`, `device-name`, `model`, `reg` and Chapter 5 “Properties” for more information.

pwd

stack:	(--)
code:	none

Displays the device-path that names the active package.

quit

stack:	(--) (R: … --)
code:	none

Aborts program execution.

r> “are from”

stack:	(-- x) (R: x --)
code:	31

Removes *x* from the return stack and places it on the stack. See `>r` for restrictions on the use of this word.

For example:

```
: copyout  ( buf addr len -- len )  >r swap r@ move r> ;
```

r@ “are fetch”

stack:	(-- x) (R: x -- x)
code:	32

Places a copy of the top of the return stack on the stack.

For example:

```
: copyout  ( buf addr len -- len )  >r swap r@ move r> ;
```

See `>r` for more details.

.r “dot are”

stack:	(n size --)
code:	9E

Converts *n* using the value of `base` and then displays it right-aligned in a field *size* digits wide. Displays a leading minus sign if *n* is negative. A trailing space is *not* displayed.

If the number of digits required to display *n* is greater than *size*, displays all the digits required with no leading spaces in a field as wide as necessary.

For example:

```
: formatted-output ( -- )
   my-length h# 8 .r  ."  length" cr
   my-width  h# 8 .r  ."  width" cr
   my-depth  h# 8 .r  ."  depth" cr
;
```

>r “to are”

stack: (x --) (R: -- x)
code: 30

Removes *x* from the stack and places it on the top of the return stack.

The return stack is a second stack, occasionally useful as a place to temporarily place numeric parameters i.e. to “get them out of the way” for a little while. For example:

```
: encode-intr  ( int-level vector -- )
   >r sbus-intr>cpu encode-int  r> encode-int  encode+
;
```

However, since the return stack is also used by the system for transferring control from word to word (and by `do` loops), improper use of `>r` or `r>` is guaranteed to crash your program. Some restrictions that *must* be observed are:

- All values placed on the return stack within a colon definition must be removed before the colon definition is exited by normal termination, `exit` or `throw`, or else the program will crash.
- No values from the return stack should be removed from within a colon definition unless they were placed there within that definition.
- Entering a `do` loop automatically places values onto the return stack. Therefore,
  - Values placed on the return stack before the loop was started will not be accessible from within the loop.
  - Values placed on the return stack within the loop must be removed before `loop`, `+loop`, or `leave` is encountered.
  - The loop indices `i` or `j` will no longer be valid when additional values have been placed on the return stack within the loop.

"ranges"

Buses such as SBus and VMEbus, whose children can be accessed with CPU load and store operations (as opposed to buses such as SCSI or IPI, whose children are accessed with a command protocol) require a way to define the bus-specific relationship between the physical address spaces of the parent and child nodes. The `"ranges"` property provides this capability.

The value of the `"ranges"` property describes the correspondence between the physical address space defined by a bus node (the “child address space”) and the physical address space of that bus node’s parent (the “parent address space”).

For a detailed description, see "ranges" on page 78.

rb! “are bee store”

stack: (byte addr --)
code: 231

Stores an 8-bit byte to a device register at *addr* with identical bit ordering as the input stack item. Data is stored with a single access operation and flushes any intervening write buffers, so that the data reaches its final destination before the next FCode method is executed.

For example:

```
: my-stat! ( byte --  ) my-stat rb! ;
```

rb! “are bee store”

stack: (byte addr --)

This optional User Interface function behaves identically to the FCode version of `rb!`.

rb@ “are bee fetch”

stack: (addr -- byte)
code: 230

Fetches *byte* from the device register at *addr*. Data is read with a single access operation. The result has identical bit ordering as the original register data.

For example:

```
: my-stat@ ( -- byte ) my-stat rb@ ;
```

rb@ “are bee fetch”

stack: (addr -- byte)

This optional User Interface function behaves identically to the FCode version of `rb@`.

read

stack: (addr len -- actual)
code: none

Reads at most *len* bytes from the device into the memory buffer beginning at *addr*. Returns *actual*, the number of bytes actually read. If *actual* is zero or negative, the read operation did not succeed.

Devices of the following types place additional requirements on their `read` method:

- network

  The `read` method receives (non-blocking) a network packet placing at most the first *len* bytes into memory at *addr*, returning either the number of bytes actually received (not placed into memory) or -2 if no packet is currently available.

- serial

  The `read` method receives a number of bytes equal to the minimum of *len* and the number of bytes available for immediate reception from the device, and places those bytes in memory at *addr*, returning either the number of bytes actually read or -2 if no bytes are currently available from the device.

For some devices, the `seek` method sets the position for the next `read`.

read-blocks

stack: (addr block# #blocks -- #read)
code: none

Reads *#blocks* records of length `block-size` bytes from the device (starting at device block *block#*) into memory (starting at *addr*). Returns *#read*, the number of blocks actually read.

If the device is not capable of random access (e.g. a sequential access tape device), *block#* is ignored.

recurse

stack: (… -- ???)
code: none

Compiles a recursive call to the command being compiled.

recursive

stack: (--)
code: none

Makes the current definition visible for a recursive call.

Normally, when a colon definition is being compiled, its name is not visible in the dictionary until the definition is completed. That way a call to that same name finds the previous version of a definition, not the one in progress. `recursive` makes the current definition visible so that subsequent uses of its name compile recursive calls to itself.

reg

stack: (phys.lo … phys.hi size --)
code: 116

This is a shorthand word for declaring the `"reg"` property on buses whose `#size-cells` property is one. Typical usage for an SBus device:

```
my-address 40.0000 +  my-space  20  reg
```

This declares that the device registers are located at offset `40.0000` through `40.001f` in this slot. The following code would accomplish the same thing:

```
my-address 40.0000 +  my-space encode-phys
20 encode-int encode+
" reg" property
```

Note that if you need to declare more than one block of register addresses or if the parent's `#size-cells` property is not equal to one, `encode-phys`, `encode-int` and `encode+` must be used repeatedly to build the prop-encoded array that is passed to the `property` method to create the `"reg"` property.

For example, `reg` cannot be used to create the `"reg"` property for PCI devices since at least three entries are required for PCI devices and since `#size-cells` is two for PCI.

See also: `property`, `"reg"` in Chapter 5 "Properties".

"reg"

This standard property specifies the range of addressable regions on the device. The `"reg"` property represents the physical address, within its parent node's address space, of the device associated with the node and also the amount of physical address space consumed by that device. In general, the `"reg"` property of a node can contain several *phys.lo … phys.hi size* specifications representing several disjoint ranges of physical address space.

In the specific case of PCI, *phys.lo … phys.hi* for the PCI Configuration Space header can be generated with `my-address` and `my-space`, and *size* is always zero. For other addressable regions, *phys.hi* must be modified to include the number of the associated base address register, the type of memory space (i.e. memory or I/O) and any other relevant information defined for *phys.hi* by the *PCI Bus Binding to IEEE Standard 1275-1994*.

As specified in the binding, the order of the pairs should be:

- An entry describing the Configuration Space for the device.
- An entry for each active base address register (BAR), in Configuration Space order, describing the entire space mapped by that BAR.
- An entry describing the Expansion ROM BAR, if the device has an Expansion ROM.
- An entry for each non-relocatable addressable resource.

For example, to declare a PCI device with:

- A register field at 10.0000-10.00ff in memory space that is controlled by the first 32-bit base address register.
- A register field at 20.0000-20.037f in I/O space that is controlled by the second 32-bit base address register.
- A 128Kbyte PCI Expansion ROM.
- A non-relocatable field at 0-fff in I/O space.

use the following:

```
my-address my-space encode-phys                     \ Config space regs
0 encode-int encode+ 0 encode-int encode+
my-address 10.0000 0 d+ my-space 0200.0010 or      \ Memory space
encode-phys encode+                                 \ BAR at 0x10
0 encode-int encode+ 100 encode-int encode+
my-address 20.0000 0 d+ my-space 0100.0014 or    \ I/O space
encode-phys encode+                              \ BAR at 0x14
0 encode-int encode+ 380 encode-int encode+
my-address my-space h# 0200.0030 or              \ PCI Expansion ROM
encode-phys encode+                              \ memory space
0 encode-int encode+ h# 2.0000 encode-int encode+
my-address my-space h# 8100.0000 or              \ Non-relocatable
encode-phys encode+                              \ memory space
0 encode-int encode+ h# 1000 encode-int encode+
" reg" property
```

For a detailed description, see "reg" on page 81.

.registers

stack: (--)
code: none

Displays the register values that were in effect when the program state was saved (i.e. when the program was suspended). The exact set of registers displayed, and the format, is system-dependent.

"relative-addressing"

The presence of this standard property indicates that each device node address is relative (i.e. local to the address space defined by the node's parent). The absence of the property indicates that device node's addresses are absolute addresses within the system-wide address space.

release

stack: (virt len --)
code: none

Free (release) addressable resource.

Frees *len* bytes of the addressable resource managed by the package containing this method, beginning at the address *virt*, making it available for subsequent use.

See also: `claim`, `alloc-mem`, `"available"`, `free-mem`

remove-abort

stack: (--)
code: none

Instructs the device driver to cease periodic polling for a keyboard abort sequence. Executed when the console input device is changed from this device to another.

repeat

stack: (C: orig-sys dest-sys --)
(--)
generates: `bbranch -offset b(>resolve)`

Terminates a `begin`...`while`...`repeat` conditional loop. See `while` for more details.

reset

stack: (--)
code: none

Puts this device into a quiescent state. The definition of “quiescent” is device-specific. This method is used primarily for permanently-installed devices (which are therefore not probed) that do not automatically assume a quiescent state after a system reset.

The `reset` method is not invoked by any standard Open Firmware functions, but may be explicitly executed for particular “problem” devices in particular Open Firmware implementations.

reset-all

stack: (--)
code: none

Reset the machine as if a power-on reset had occurred.

This command is used to initiate a system power-on reset, thus re-initializing the hardware state and Open Firmware’s data structures as if a power-on reset had occurred.

reset-screen

stack: (--)
code: 158

`reset-screen` is one of the `defer` words of the display device interface. The terminal emulator package executes `reset-screen` when it has processed a character sequence that calls for resetting the display device to its initial state. `reset-screen` puts the display device into a state in which display output is visible (e.g. enable video).

This word is initially empty, but *must* be loaded with an appropriate routine in order for the terminal emulator to function correctly. This can be done with `to`, or it can be loaded automatically with `fb1-install` or `fb8-install` (which load `fb1-reset-screen` or `fb8-reset-screen`, respectively). These words are NOPs, so it is very common to first call `fbx-install` and then to override the default setting for `reset-screen` with:

```
['] my-video-on   to reset-screen
```

restore

stack: (--)
code: none

Restores a device to a usable state after an unexpected reset.

On some systems, unexpected system errors result in a bus reset that turns off some devices, but does not necessarily destroy the machine state necessary for debugging the error. In such systems, the system-dependent firmware handler for that reset condition may execute the `restore` methods of the console input and output devices, in order to re-enable those devices for user interaction and subsequent debugging.

resume

stack: (--)
code: none

`resume` is one of the source-level debugger control words. After the “f” keystroke is used with the stepper to enter a “subordinate interpreter”, `resume` is used to exit back to the stepper.

return

stack: (--)
code: none

`return` is one of the breakpoint commands. After a breakpoint has been encountered within a subroutine, `return` can be used to continue execution until the return from the subroutine.

ring-bell

stack: (--)
code: none

Causes the device to emit a brief audible sound (beep).

See also: `blink-screen`

rl! “are el store”

stack: (quad qaddr --)
code: 235

Stores a 32-bit word to a device register at *qaddr* with identical bit ordering as the input stack item. *qaddr* must be 32-bit aligned. Data is stored with a single access operation and flushes any intervening write buffers, so that the data reaches its final destination before the next FCode method is executed.

For example:

```
: my-reg! ( n -- ) my-reg rl! ;
```

rl! “are el store”

stack: (quad qaddr --)

This optional User Interface function behaves identically to the FCode version of `rl!`.

rl@ “are el fetch”

stack: (qaddr -- quad)
code: 234

Fetches a 32-bit word from the device register at *qaddr. qaddr* must be 32-bit aligned. Data is read with a single access operation. The result has identical bit ordering as the original register data.

For example:

```
: my-reg@ ( -- n ) my-reg rl@ ;
```

rl@

"are el fetch"

stack: (qaddr -- quad)

This optional User Interface function behaves identically to the FCode version of rl@.

roll

stack: (xu … x1 x0 u -- xu-1 … x1 x0 xu)

code: 4F

Removes the *u*+1-th stack value, not including u itself, where the remaining stack items have indices beginning with 0. The *u*-th stack item is then placed on the top of the stack, moving the remaining items down one position. *u* must be between 0 and two less than the total number of elements on the stack (including *u*).

```
0 roll is a null operation
1 roll is equivalent to swap    ( n1 n2 -- n2 n1 )
2 roll is equivalent to rot     ( n1 n2 n3 -- n2 n3 n1 )
3 roll is equivalent to         ( n1 n2 n3 n4 -- n2 n3 n4 n1 )
```

For the sake of readability and performance, minimize your use of `roll`.

rot

"rote"

stack: (x1 x2 x3 -- x2 x3 x1)

code: 4A

Rotates the top three stack entries, bringing the deepest to the top.

-rot

"minus rote"

stack: (x1 x2 x3 -- x3 x1 x2)

code: 4B

Rotates the top three stack entries in the direction opposite from `rot`, putting the top number underneath the other two.

2rot

"two rote"

stack: (x1 x2 x3 x4 x5 x6 -- x3 x4 x5 x6 x1 x2)

code: 56

Rotates the top three pairs of numbers, bringing the third pair to the top of the stack.

rshift

stack: (x1 u -- x2)

code: 28

Shifts *x1* right by *u* bit-places while zero-filling the high bits.

rw!

"are double you store"

stack: (w waddr --)

code: 233

Stores a 16-bit word to a device register at *waddr* with identical bit ordering as the input stack item. *waddr* must be 16-bit aligned. Data is stored with a single access operation and flushes any intervening write buffers, so that the data reaches its final destination before the next FCode method is executed.

For example:

```
: my-count! ( w -- ) my-count rw! ;
```

rw! “are double you store”

stack: (w waddr --)

This optional User Interface function behaves identically to the FCode version of `rw!`.

rw@ “are double you fetch”

stack: (waddr -- w)
code: 232

Fetches a 16-bit word from the device register at *waddr*. *waddr* must be 16-bit aligned. Data is read with a single access operation. The result has identical bit ordering as the original register data.

For example:

```
: my-count@ ( -- w ) my-count rw@ ;
```

rw@ “are double you fetch”

stack: (waddr -- w)

This optional User Interface function behaves identically to the FCode version of `rw@`.

rx! “are ecks store”

stack: (o oaddr --)
code: 22F

Stores a 64-bit word to a device register at *oaddr* with identical bit ordering as the input stack item. *oaddr* must be 64-bit aligned. Data is stored with a single access operation and flushes any intervening write buffers, so that the data reaches its final destination before the next FCode method is executed.

This function is only available on 64-bit implementations.

rx! “are ecks store”

stack: (o oaddr --)

This optional User Interface function behaves identically to the FCode version of `rx!`.

This function is only potentially available on 64-bit implementations.

rx@ “are ecks fetch”

stack: (oaddr -- o)
code: 22E

Fetches a 64-bit word from the device register at *oaddr. oaddr* must be 64-bit aligned. Data is read with a single access operation. The result has identical bit ordering as the original register data.

This function is only available on 64-bit implementations.

rx@ “are ecks fetch”

stack: (oaddr -- o)

This optional User Interface function behaves identically to the FCode version of `rx@`.

This function is only potentially available on 64-bit implementations.

s"

stack: ([text<">] -- text-str text-len)
generates: `b(") len-byte xx-byte xx-byte … xx-byte`

Gather the immediately-following string delimited by `"` . Return the location of the string *text-str text-len*.

Since an implementation is only required to provide two temporary buffers, a program cannot depend on the system’s ability to simultaneously maintain more than two distinct interpreted strings. Compiled strings do not have this limitation, since they are not stored in the temporary buffers.

s.

stack: (n --)
generates: `(.) type space`

Displays the absolute value of *n* in a free-field format according to the current value of `base`. Displays a trailing space and, if *n* is negative, a leading minus sign. Even if the base is hexadecimal, *n* will be printed in signed format

See also: `.`

#s

stack: (ud -- 0 0)
code: C8

Converts the remaining digits in pictured numeric output.

.s

stack: (… -- …)
code: 9F

Displays the contents of the data stack (using .) according to `base`. The top of the stack appears on the right. The contents of the stack are unchanged.

For example:

```
ok 1 2 3 .s
1 2 3
ok . . .
3 2 1
```

sbus-intr>cpu

stack: (sbus-intr# -- cpu-intr#)
code: 131

Convert the SBus interrupt level (1-7) to the CPU interrupt level. The mapping performed will be system-dependent.

This word is included because of the possibility that, even on systems that nominally do not support SBus, SBus devices might be used via a bus-to-bus bridge.

screen

A device alias. If the value of `screen` is not previously specified, the system will create `screen` using as its value the path of the first device of type `display` that was found during the probing process. If the `output-device` configuration variable is set to `screen`, this process results in auto-configuration of the console output device.

`screen` is the suggested default value for the `output-device` configuration variable.

screen-#columns

stack: (-- n)
code: none

This configuration variable is an integer specifying the maximum number of columns on the console output device. Standard display packages use this value to determine the width in characters of their text region. If the device is incapable of displaying that many columns, the device restrictions prevail.

The suggested default value of `screen-#columns` is 80.

screen-height

stack: (-- height)
code: 163

A `value`, containing the total height of the display (in pixels). It can also be interpreted as the number of “lines” of memory.

`screen-height` is an internal value used by the `fb1-` and `fb8-` frame buffer support packages. In particular, this value is used in `fbx-invert`, `fbx-erase-screen`, `fbx-blink-screen` and in calculating `window-top`. `fb1-install` and `fb8-install` set it to the value of their height argument.

This function is included for historical compatibility. There is little reason for an FCode Program to use it. In fact, “standard” FCode Programs are forbidden from altering its value directly.

screen-#rows

stack: (-- n)
code: none

This configuration variable is an integer specifying the maximum number of rows on the console output device. Standard display packages use this value to determine the height in text lines of their text region. If the device is incapable of displaying that many rows, the device restrictions prevail.

The suggested default value of `screen-#rows` is 24.

screen-width

stack: (-- width)
code: 164

A `value`, containing the width of the display (in pixels). It can also be interpreted as the number of pixels (in memory) between one screen location and the next location immediately below it. The latter definition takes precedence if there is a conflict (e.g. there are unused/invisible memory locations at the end of each line).

`screen-width` is an internal value used by the `fb1-` and `fb8-` frame buffer support packages. `fb1-install` and `fb8-install` set it to their width argument.

This function is included for historical compatibility. There is little reason for an FCode Program to use it. In fact, "standard" FCode Programs are forbidden from altering its value directly.

s>d

stack: (n -- d)
code: none

Converts a number to a double number.

security-#badlogins

stack: (-- n)
code: none

This configuration variable is an integer containing the total count of invalid security access attempts. This counter is incremented by one, whenever a bad password is entered when attempting to enter the command interpreter while `security-mode` is set (to either `command` mode or `full` mode).

The value in `security-#badlogins` is not affected by the `set-default` or `set-defaults` commands.

There is no suggested default value for `security-#badlogins`.

security-mode

stack: (-- n)
code: none

This configuration variable contains the level of security access protection. When security is in effect, user knowledge of a password is required to allow use of most commands through the command interpreter.

The following keywords denote the security levels:

Table 38 `security-mode` Settings

Keyword	Description
none	No security, no password required.
command	Requires password entry to execute any command except for go, boot (default device and default file), or automatic boot after system power-on or boot call.
full	Requires password entry to execute any command except for go command. Automatic booting is disabled, machine will not automatically re-boot after a power failure.

For example:

```
ok setenv security-mode full
```

The value of security-mode is not affected by the `set-default` or `set-defaults` commands.

There is no suggested default value for `security-mode`.

It is not possible to determine the level of security protection from within a program, as the value *n* returned by this command cannot be related unambiguously to the security level keywords.

security-password

stack: (-- password-str password-len)
code: none

The value of this configuration variable is a string specifying the security password text string. The value of `security-password` is normally set with the `password` command, although `setenv` can also be used.

The value of `security-password` is not be displayed when `printenv` is executed. The value of `security-password` is not affected by the `set-default` or `set-defaults` commands.

There is no suggested default value for `security-password`.

see

stack: ("old-name< >" --)
code: none

Decompiles the Forth command *old-name*.

For example:

```
ok see see
: see
   ' ['] (see) catch if
      drop
   then
;
```

(see)

stack: (xt --)
code: none

Decompiles the Forth command whose execution token is *xt*.

For example:

```
ok ['] see (see)
: see
   ' ['] (see) catch if
      drop
   then
;
```

seek

stack: (pos.lo pos.hi -- status)
code: none

Sets the device position at which the next read or write will take place. The position is specified by a pair of numbers *pos.lo pos.hi*, whose interpretation depends on the device type. *status* is -1 if the operation fails and either zero or one if it succeeds.

select

stack: ("device-specifier< >" --)
code: none

A User Interface extension provided by some implementations (e.g. FirmWorks/Sun).

Creates an instance chain for the device specified by *device-specifier.*

See also: “Using select” on page 37.

select-dev

stack: (dev-str dev-len --)
code: none

A User Interface extension provided by some implementations (e.g. FirmWorks/Sun).

Creates an instance chain for the device specified by *dev-str dev-len*.

See also: “Using select-dev” on page 35.

selftest

stack: (-- 0 | error-code)
code: none

Note – United States Patent No. 4,633,466, "Self Testing Data Processing System with Processor Independent Test Program", issued December 30, 1986 may apply to some or all elements of Open Firmware selftest. Anyone implementing Open Firmware should take such steps as may be necessary to avoid infringement of that patent and any other applicable intellectual property rights.

Performs the selftest for this device. Returns 0 if successful or a device-specific nonzero *error-code* if an error is detected. The complexity of this test will typically be much greater than that of the test performed when `open` is called.

This method is typically invoked by the user commands `test` or `test-all`, via `execute-device-method`. Consequently, the package's `open` method has not necessarily been executed before `selftest` is invoked. (`execute-device-method` does not call `open`, but it is possible for the device to have already been previously opened.) `selftest` should leave the device in a state similar to that before `selftest` was executed. Therefore, `selftest` is responsible for establishing any device state necessary to perform its function prior to starting the tests and for releasing any resources that were allocated during the process after completing the tests.

The extent of the testing performed by `selftest` can be made to be dependent upon the value returned by `diagnostic-mode?`; if so, more extensive testing should be performed when `diagnostic-mode?` return `true`.

selftest-#megs

stack:	(-- n)
code:	none

This configuration variable is an integer specifying the maximum number of megabytes of memory that should be tested by the `selftest` routine of the `"memory"` node (i.e. the node whose device-path is `/memory`). In most systems that memory test is automatically executed after the secondary diagnostics. (Some smaller portion of memory is usually tested by POST, as well.) `selftest-#megs` controls the extent of memory selftest. If `diagnostic-mode?` is `true`, the system may ignore the value of `selftest-#megs`.

The suggested default value of `selftest-#megs` is 1.

"serial"

This is the standard property value of the `"device_type"` property for byte-oriented devices such as a serial port.

Devices of type `"serial"` must implement the following methods:

- `open`
- `close`
- `read`

  The `read` method receives a number of bytes equal to the minimum of *len* and the number of bytes available for immediate reception from the device, and places those bytes in memory at *addr*, returning either the number of bytes actually read or -2 if no bytes are currently available from the device.

- `write`
- `install-abort`
- `remove-abort`

If a device of type `"serial"` can cause the display to become invisible (e.g. the video is turned off) in the case of an unexpected system reset, and if the display can be restored to visibility without performing memory mapping or memory allocation operations, the package should implement the `restore` method.

If a device of type `"serial"` has an audible annunciator that is activated by some action other than sending an ASCII BEL character then the package should implement the `ring-bell` method.

A package with this `"device_type"` property value may implement additional device-specific methods.

See also: `character-set`

set-args

stack: (arg-str arg-len unit-str unit-len --)
code: 23F

Sets the address and arguments of a new device node.

unit-string is a text string representation of a physical address within the address space of the parent device. `set-args` translates `unit-string` to the equivalent numerical representation by executing the parent instance's `decode-unit` method, and sets the current instance's probe-address (i.e. the values returned by `my-address` and `my-space`) to that numerical representation.

`set-args` then copies the string *arg-string* to instance-specific storage, and arranges for `my-args` to return the address and length of that copy when executed from the current instance.

`set-args` is typically used just after `new-device`. `new-device` creates and selects a new device node, and `set-args` sets its probe-address and arguments. Subsequently, the device node's properties and methods are created by interpreting an FCode Program with `byte-load` or by interpreting Forth source code.

The empty string is commonly used as the arguments for a new device node. For example:

```
0 0 " 3" set-args
```

set-default

stack: ("param-name<eol>" --)
code: none

Sets the specified configuration variable to its default value.

For example:

```
ok set-default auto-boot?
```

Some configuration variables are unaffected by `set-default`, as noted in individual configuration variable command descriptions.

set-defaults

stack: (--)
code: none

Resets most configuration variables to their default values.

Some configuration variables are unaffected by `set-defaults`, as noted in individual configuration variable command descriptions.

setenv

stack: ("nv-param< >new-value<eol>" --)
code: none

Sets the configuration variable *nv-param* to the value specified by *new-value*.

If *new-value* is the empty string, `setenv` displays an error message and returns.

Otherwise, it performs the equivalent of `$setenv` with string arguments denoting *nv-param* and *new-value*.

For example:

```
ok setenv auto-boot? true
ok setenv oem-banner The nEw TeXt looks  Just like this!
```

See also: `$setenv`

$setenv

stack: (data-addr data-len name-str name-len --)
code: none

Sets the configuration variable *name-string* to the value specified by *data-addr data-len*. `$setenv` interprets the new value according to the configuration variable's configuration variable data type. If the given value string is not suitable for that data type, `$setenv` displays an error message. Otherwise, `$setenv` sets the configuration variable to the new value, truncating it to fit the available space (if necessary), and then displays the new value.

For example:

```
ok " new-value" " nv-name" $setenv
```

See also: `setenv`

set-font

stack: (addr width height advance min-char #glyphs --)
code: 16B

This routine declares the font table to be used for printing characters on the screen. This routine *must* be called if you wish to use *any* of the `fb1-` or `fb8-` utility routines or `>font`.

Normally, `set-font` is called just after `default-font`. `default-font` leaves the exact set of parameters needed by `set-font` on the stack. This approach allows your FCode Program to inspect and/or alter the default parameters if desired. See `default-font` for more information on these parameters.

set-rev-level

stack: (revision --)
generates: value of "revision-level" field of PCI Expansion ROM header

A FirmWorks extension to the tokenizer. `set-rev-level` sets the value used by `pci-header-end` when creating the "revision level" field of the PCI Expansion ROM. `set-rev-level` must be executed prior to `pci-header`.

See also: `fcode-end`, `pci-header`, `pci-header-end`, `not-last-image`, `set-vpd-offset`

set-vpd-offset

stack: (addr --)
generates: value of “pointer to vital product data” field of PCI Expansion ROM header

A FirmWorks extension to the tokenizer. `set-vpd-offset` sets the value used by `pci-header-end` when creating the “pointer to vital product data” field of the PCI Expansion ROM. `set-vpd-offset` must be executed prior to `pci-header`.

See also: `fcode-end`, `pci-header`, `pci-header-end`, `not-last-image`, `set-rev-level`

set-token

stack: (xt immediate? fcode# --)
code: DB

Assigns the FCode number *fcode#* to the FCode function whose execution token is *xt*, with compilation behavior specified by *immediate?* as follows:

- If *immediate?* is zero, then the FCode Evaluator will execute the function’s execution semantics if it encounters that FCode number while in interpretation state, or append those execution semantics to the current definition if it encounters that FCode number while in compilation state.
- If *immediate?* is nonzero, the FCode Evaluator will execute the functions’s FCode Evaluation semantics anytime it encounters that FCode number.

show-devs

stack: ("{device-specifier}<eol>" --)
code: none

Shows the full device path for each device in the sub-tree of the device tree underneath the node specified by *device-specifier.*

If *device-specifier* is the empty string (i.e. there is nothing on the command line following `show-devs`), shows all system devices.

showstack

stack: (… -- …)
code: none

Displays the entire stack, with a format similar to the `.s` command, just before each `ok` prompt.

This feature can be turned off with the `noshowstack` command. The system default is `noshowstack`.

See also: `noshowstack`

$sift

stack: (text-addr text-len --)
code: none

Display all command-names containing *text-string*.

Searches the current vocabulary and displays the names of all commands which include *text-string* as part of the command-name. Upper and lower-case distinctions are ignored. This command is useful for finding all commands of a particular “type” or for finding any command where the name is only partially known.

For example:

```
ok " spaces" $sift

               In vocabulary hidden
(1e10e90) .spaces
               In vocabulary forth
(1e0d990) spaces         (1e0302c) backspaces
(1e02fec) spaces
```

See also: `sifting` `order`

sifting

stack: ("text< >" --)
code: none

Display all command-names containing *text*.

For example:

```
ok sifting spaces

               In vocabulary hidden
(1e10e90) .spaces
               In vocabulary forth
(1e0d990) spaces         (1e0302c) backspaces
(1e02fec) spaces
```

See also: `$sift`

sign

stack: (n --)
code: 98

If *n* is negative, appends an ASCII “-” (minus sign) to the pictured numeric output string. Typically used between <# and #>. See `(.)` for a typical usage.

### "#size-cells"

This standard property applies to bus nodes. The property value is an integer encoded with `encode-int` and specifies the number of cells that are used to encode the size field of a child’s `"reg"` format property. A missing `"#size-cells"` property signifies the default value of one. Plug-in devices should use the value specified for their bus and, if unspecified, should use the default value of one.

For PCI, `"#size-cells"` is 2 which reflects PCI’s 64-bit address space.

sm/rem

stack: (d n -- rem quot)
code: none

Divides *d* by *n* returning *rem* and *quot*. *rem* carries the sign of *d* or is zero. *quot* is the quotient rounded towards zero.

source

stack: (-- addr len)
code: none

Returns the location and size of the input buffer.

space

stack: (--)
generates: `bl emit`

Displays a single ASCII space character.

spaces

stack: (cnt --)
generates: `0 max 0 ?do space loop`

Displays *cnt* ASCII space characters. Nothing is displayed if *cnt* is zero.

span

stack: (-- a-addr)
code: 88

A `variable` containing the count of characters actually received and stored by the last execution of `expect`.

For example:

```
h# 10 buffer: my-name-buff
: hello ( -- )
   ." Enter Your First name " my-name-buff h# 10 expect
   ." FirmWorks Welcomes " my-name-buff span @ type cr
;
```

start0

stack: (--)
code: F0

`start0` may only be used as the first byte of an FCode Program. `start0`:

- Sets the `spread` value to 0 causing the FCode Evaluator to read successive bytes of the current FCode Program from the same address.
- Establishes the use of 16-bit branches.
- Reads an FCode header from the current FCode Program and either discards it or uses it to verify the integrity of the current FCode program in an implementation-dependent manner.

For portability, the preferred way to begin an FCode program in source form is with the `fcode-version2` tokenizer macro. This macro causes the tokenizer to begin the FCode binary with the appropriate `start` byte and an FCode header.

See also: `fcode-version2`, `start1`, `start2`, `start4`, `version1`

start1

stack: (--)
code: F1

`start1` may only be used as the first byte of an FCode Program. `start1`:

- Sets the `spread` value to 1 causing the FCode Evaluator to read successive bytes of the current FCode Program from addresses one address unit apart.
- Establishes the use of 16-bit branches.
- Reads an FCode header from the current FCode Program and either discards it or uses it to verify the integrity of the current FCode program in an implementation-dependent manner.

For portability, the preferred way to begin an FCode program in source form is with the `fcode-version2` tokenizer macro. This macro causes the tokenizer to begin the FCode binary with the appropriate `start` byte and an FCode header.

See also: `fcode-version2`, `start0`, `start2`, `start4`, `version1`

start2

stack: (--)
code: F2

`start2` may only be used as the first byte of an FCode Program. `start2`:

- Sets the `spread` value to 2 causing the FCode Evaluator to read successive bytes of the current FCode Program from addresses two address units apart.
- Establishes the use of 16-bit branches.
- Reads an FCode header from the current FCode Program and either discards it or uses it to verify the integrity of the current FCode program in an implementation-dependent manner.

For portability, the preferred way to begin an FCode program in source form is with the `fcode-version2` tokenizer macro. This macro causes the tokenizer to begin the FCode binary with the appropriate `start` byte and an FCode header.

See also: `fcode-version2`, `start0`, `start1`, `start4`, `version1`

start4

stack: (--)
code: F3

`start4` may only be used as the first byte of an FCode Program. `start4`:

- Sets the `spread` value to 4 causing the FCode Evaluator to read successive bytes of the current FCode Program from addresses four address units apart.
- Establishes the use of 16-bit branches.
- Reads an FCode header from the current FCode Program and either discards it or uses it to verify the integrity of the current FCode program in an implementation-dependent manner.

For portability, the preferred way to begin an FCode program in source form is with the `fcode-version2` tokenizer macro. This macro causes the tokenizer to begin the FCode binary with the appropriate `start` byte and an FCode header.

See also: `fcode-version2`, `start0`, `start1`, `start2`, `version1`

state

stack: (-- a-addr)
code: DC

A `variable` containing *true if* the system is in compilation state.

state-valid

stack: (-- a-addr)
code: none

A `variable` containing *true if* `saved-program-state` is valid. `state-valid` is set to *true by* `init-program` and by actions that result in the saving of program state.

`saved-program-state` must be valid in order for execution with `go` to perform properly.

status

stack: (--)
code: none

`status` is a `defer` word, initially vectored to `noop`, which can be used to modify the user interface prompt to display whatever additional information the user wishes to see.

For example:

```
ok hex : showbase  ( -- )  ." ( " base @ .d ." ) " ;
ok ['] showbase to status
( 16 ) ok ['] noop to status
ok
```

"status"

If this standard property is present, the value is a string indicating the status of the device, as follows:

Table 39 "status" Property Value Descriptions

"status"	Description
"okay"	The device is believed to be operational.
"disabled"	The device represented by this node is not operational, but it might become operational in the future (e.g. an external switch is turned off, or something isn't plugged in.)
"fail"	The device represented by this node is not operational because a fault has been detected, and it is unlikely that the device will become operational without repair. No additional failure details are available.

Table 39 (Continued)`"status"` Property Value Descriptions

`"status"`	**Description**
"fail-xxx"	The device represented by this node is not operational because a fault has been detected, and it is unlikely that the device will become operational without repair. "xxx" is additional human-readable information about the particular fault condition that was detected.

The absence of the `"status"` property means that the operational status is unknown or okay.

stdin

stack: (-- a-addr)
code: none

A `variable` containing the ihandle of the console input device.

"stdin"

This property appears in the `/chosen` node. The property value is an integer encoded with encode-int containing the ihandle of the console input device.

stdout

stack: (-- a-addr)
code: none

A `variable` containing the ihandle of the console output device.

"stdout"

This property appears in the `/chosen` node. The property value is an integer encoded with encode-int containing the ihandle of the console output device.

step

stack: (--)
code: none

`step` is one of the breakpoint commands. After a breakpoint has been encountered, `step` resumes client program execution as with go, but only executes one instruction. The effect is as if breakpoints were established at the possible successors to that instruction and then automatically removed when the breakpoint is handled.

.step

stack: (--)
code: none

`.step` is a `defer` word that is executed whenever a single step occurs. The default behavior is `.instruction`.

For example, to display registers at every single step:

```
ok ['] .registers to .step
```

See also: `defer`

stepping

stack: (--)
code: none

Sets "step mode" for Forth source-level debugging. This mode allows interactive step-by-step execution of the command being debugged. "Step mode" is the default.

While in "step mode", before the execution of each command called by the debugged command, the user is prompted for one of a number of keystrokes. See `debug` for a list of these keystrokes.

steps

stack: (n --)
code: none

Executes `step` *n* times.

struct

stack: (-- 0)
generates: `0`

Initializes a `struct`...`field` structure by leaving a zero on the stack to define the initial offset. See `field` for details.

suppress-banner

stack: (--)
code: none

If executed within the NVRAM script, suppresses the automatic execution of the following Open Firmware start-up sequence:

- `probe-all`
- `install-console`
- `banner`

`suppress-banner` is useful for modifying the standard startup sequence. For a usage example, see "Patching FCode of a Plug-in Card" on page 22.

See also: `banner`, `oem-banner`, `oem-banner?`, `oem-logo`, `oem-logo?`, `probe-all`

suspend-fcode

stack: (--)
code: 215

Tells the FCode interpreter that the device identification properties for the active package have been declared, and that the interpreter may postpone interpreting the remainder of the package if it so chooses.

If the FCode interpreter postpones (suspends) interpretation, it saves the state of the interpretation process so that interpretation can continue later. Attempts to open a suspended package cause the FCode interpreter to resume and complete the interpretation of that package before executing the package's `open` method.

For example:

```
fcode-version2
   " INTL,my-name" namea
   " INTL,my-model" encode-string " model" property
   suspend-fcode
   …
fcode-end
```

This feature is intended to save memory space and reduce the system start-up time by preventing the compilation of FCode drivers that are not actually used.

swap

stack:	(x1 x2 -- x2 x1)
code:	49

Exchanges the top two stack items.

2swap

stack:	(x1 x2 x3 x4 -- x3 x4 x1 x2)
code:	55

Exchanges the top two pairs of stack items.

sym

stack:	("name< >" -- n)
code:	none

Returns the value of the client program symbol *name*. If *name* cannot be resolved to a symbol, `sym` performs an `abort`. Otherwise, `sym` returns the symbol value *n* corresponding to *name*.

sym>value

stack:	(addr len -- addr len false | n true)
code:	none

This `defer` word is executed when the symbolic debugger needs to translate a symbol name into its corresponding value. If `sym>value` is present, the Forth interpreter attempts to perform such translation if a word is neither found in the normal dictionary search nor recognized as number. The translation is also attempted by `sym`.

If a symbol whose name matches the string given by *addr len* is defined, `sym>value` returns the corresponding symbol value and *true*. Otherwise, `sym>value` returns its *addr len* arguments and *false*.

The default action for `sym>value`, when no symbol table is present, is to return *addr len* and *false*. A program can provide a symbol table and install a command into `sym>value` with `to` to search that symbol table.

See also: `value>sym`

sync

stack: (--)
code: none

Flushes the system file buffers after a program interrupt.

Equivalent to: `callback sync` <eol>

The suggested callback behavior of the `sync` command is to flush the system's file buffers. `sync` is often used after a client program has been forcibly interrupted by aborting to Open Firmware.

test

stack: ("device-specifier<eol>" --)
code: none

If the device node specified by *device-specifier* has a `selftest` method, `test` invokes it with `execute-device-method`. Otherwise, `test` displays an error message.

For example:

```
ok test device-alias
```

test-all

stack: ("{device-specifier}<eol>" --)
code: none

`test-all` visits each node in the sub-tree of the device tree at and below the specified node, or below the root node if no node is specified. For each node with a `selftest` method, `test-all` invokes that `selftest` routine using `execute-device-method`.

The system may choose not to test certain active devices that it believes are "unsafe" to test while active.

For example:

```
ok test-all device-alias
```

then

stack: (C: orig-sys --)
(--)
generates: `b(>resolve)`

Terminates an `if`…`then` or an `if`…`else`…`then` conditional structure. See `if` for more details.

throw

stack: (… error-code -- ??? error-code | …)
code: 218

If *error-code* is 0, drops *error-code* and exits.

If *error-code* is non-zero, pops the exception frame, restores the value of `my-self`, restores the input source, adjusts the stack depth as specified by the exception frame, pushes *error-code* on top of the stack, and transfers control to just beyond the `catch` that pushed the exception frame.

If *error-code* is non-zero and there is no exception frame on the exception stack, the behavior is as follows:

- If *error-code* is -1, `abort` is performed.
- If *error-code* is -2, `abort"` is performed.
- Otherwise, the system may display an implementation-dependent message giving information about the condition associated with the `throw` code *error-code*, and `abort` is performed.

See `catch` for an example of use.

till

stack: (addr --)
code: none

`till` is one of the breakpoint commands. `till` adds the specified address to the breakpoint list and resumes execution.

`till` is equivalent to: `+bp go`

to

stack: (param [old-name< >] --)
generates: `b(to)` `old-FCode#`

Changes the contents of a `value` or a `defer` word:

```
number to name  ( for a value )
xt    to name  ( for a defer word )
```

toggle-cursor

stack: (--)
code: 159

`toggle-cursor` is one of the `defer` words of the display device interface. The terminal emulator package executes `toggle-cursor` when it is about to process a character sequence that might involve screen drawing activity, and executes it again after it has finished processing that sequence. The first execution removes the cursor from the screen so that any screen drawing will not interfere with the cursor, and the second execution restores the cursor, possibly at a new position, after the drawing activity related to that character sequence is finished. `toggle-cursor` is also called once during the terminal emulator initialization sequence.

If the text cursor is on, `toggle-cursor` turns it off. If the text cursor is off, `toggle-cursor` turns it on. (On a bitmapped display, a typical implementation of this function inverts the pixels of the character cell to the right of the current cursor position.)

`toggle-cursor` is initially empty, but must be loaded with an appropriate routine in order for the terminal emulator to function correctly. This can be done with `to`, or it can be loaded automatically with `fb1-install` or `fb8-install` (which load `fb1-toggle-cursor` or `fb8-toggle-cursor`, respectively).

If the display device hardware has internal state (for example color map settings) that might have been changed by external software without firmware's knowledge, that hardware state should be re-established to the state that the firmware driver requires when the cursor is toggled to the “off” state (which indicates that firmware drawing operations are about to begin). This situation can occur, for example, when an operating system is using a display device, but that operating system uses firmware text output services from time to time, e.g. for critical warning messages.

See also: `to`, `fb1-install`, `fb8-install`

tokenizer[

stack: (--)
code: none

This is a tokenizer command that ends FCode byte generation and begins interpretation of the following text as tokenizer commands (up to the closing `]tokenizer`). A `tokenizer[`...`]tokenizer` sequence may be used anywhere in an FCode Program, either within any definition or outside of definitions.

One plausible use for `tokenizer[` would be to generate debugging text during the tokenizing process. (A `cr` flushes the text from the output buffer immediately, which is useful if the tokenizer crashes.) For example:

```
…
tokenizer[  .( step a)  cr  ]tokenizer
…
tokenizer[  .( step b)  cr  ]tokenizer
…
```

`emit-byte` can be used with `tokenizer[` to output a desired byte of FCode. This would be useful, for example, if you wished to generate a new FCode command that the tokenizer did not understand. For example:

```
…
tokenizer[  1 emit-byte  27 emit-byte  ]tokenizer
\ manually output finish-device fcode
…
```

]tokenizer

stack: (--)
code: none

Ends a tokenizer-only command sequence. See `tokenizer[`.

tracing

stack: (--)
code: none

Sets “trace mode” for Forth source-level debugging. This mode causes execution of the word being debugged to be traced, showing the name and stack contents for each command called by the debugged command. Tracing continues until `stepping` is executed or a system reset takes place.

See also: `debug`

-trailing

stack: (str len1 -- str len2)
code: none

Removes any trailing spaces from a string.

translate

stack: (virt -- false | phys.lo … phys.hi mode … true)
code: none

If a valid virtual to physical address translation exists for the virtual address *virt*, `translate` returns the physical address *phys.lo* … *phys.hi*, the translation mode *mode* … (typically, but not necessarily, one cell) and *true*. Otherwise, `translate` returns *false*. The physical address format is the same as that of the `"memory"` node (i.e. the node whose ihandle is given by the value of the /chosen node's `"memory"` property). The interpretation of *mode* … is MMU-dependent.

true

stack: (-- true)
generates: `-1`

Leaves the value for the `true` flag (which is -1) on the stack.

tuck

stack: (x1 x2 -- x2 x1 x2)
code: 4C

Copies the top stack item underneath the second stack item.

type

stack: (text-str text-len --)
code: 90

A `defer` word that transfers *text-len* characters to the output beginning with the character at address *text-str* and continuing through *text-len* consecutive addresses. No action is taken if *text-len* is zero.

For example:

```
h# 10 buffer: my-name-buff
: hello ( -- )
   ." Enter Your First name " my-name-buff h# 10 expect
   ." FirmWorks Welcomes " my-name-buff span @ type cr
;
```

The output will go either to a framebuffer or to a serial port depending on which is enabled.

u# "you number"

stack: (u1 -- u2)
code: 99

The remainder of *u1* divided by the value of `base` is converted to an ASCII character and appended to the output string with `hold`. *u2* is the quotient and is maintained for further processing. Typically used between <# and #>.

u#> "you number greater than"

stack: (u -- str len)
code: 97

Pictured numeric output conversion is ended dropping *u*. *str* is the address of the resulting output array. *len* is the number of characters in the output array. *str* and *len* together are suitable for `type`. See `(.)` and `(u.)` for typical usages.

u#s "you number ess"

stack: (u1 -- u2)
code: 9A

u1 is converted, appending each resultant character into the pictured numeric output string until the quotient is zero (see: #). A single zero is added to the output string if *u1* was initially zero. Typically used between <# and #>. See `(.)` and `(u.)` for typical usages.

This word is equivalent to calling # repeatedly until the number remaining is zero.

u* "you star"

stack: (u1 u2 -- uprod)
code: none

Multiplies *u1* by *u2* yielding *uprod*, all unsigned.

u. "you dot"

stack: (u --)
code: 9B

Displays *u* as an unsigned number in a free-field format according to the value in `base`. A trailing space is also displayed.

For example:

```
ok hex -1 u.
ffffffff
```

u<

stack: (u1 u2 -- unsigned-less?)
code: 40

Returns *true* if *u1* is less than *u2* where *u1* and *u2* are treated as unsigned integers.

u<=

stack: (u1 u2 -- unsigned-less-or-equal?)
code: 3F

Returns *true* if *u1* is less than or equal to *u2* where *u1* and *u2* are treated as unsigned integers.

u>

stack: (u1 u2 -- unsigned-greater?)
code: 3E

Returns *true* if *u1* is greater than *u2* where *u1* and *u2* are treated as unsigned integers.

u>=

stack: (u1 u2 -- unsigned-greater-or-equal?)
code: 41

Returns *true* if *u1* is greater than or equal to *u2* where *u1* and *u2* are treated as unsigned integers.

(u.)

stack: (u -- str len)
generates: `<# u#s u#>`

This is a numeric conversion primitive used to implement display words such as `u.` . It converts an unsigned number into a string according to the value in `base`.

For example:

```
ok hex d# -12 (u.) type
fffffff4
```

u2/ "you two slash"

stack: (x1 -- x2)
code: 58

x2 is the result of *x1* logically shifted right one bit. A zero is shifted into the vacated sign bit.

For example:

```
ok -2 u2/ .s
7fffffff
```

um* "you em star"

stack: (u1 u2 -- ud.prod)
code: D4

Multiplies two unsigned 32-bit numbers yielding an unsigned 64-bit product.

For example:

```
ok hex 3 3 u*x .s
9 0
ok 4 ffff.ffff u*x .s
fffffffc 3
```

See also: `um/mod`, `d+`, `d-`

um/mod “you em slash mod”

stack: (ud u -- urem uquot)
code: D5

Divides an unsigned 64-bit number by an unsigned 32-bit number yielding an unsigned 32-bit remainder and quotient

See also: `um*`, `d+`, `d-`

u/mod “you slash mod”

stack: (u1 u2 -- urem uquot)
code: 2B

rem is the remainder and *quot* is the quotient after dividing *u1* by *u2*. All values and arithmetic are unsigned. All values are 32-bit.

For example:

```
ok -1 5 u/mod .s
0 33333333
```

unaligned-l!

stack: (quad addr --)
code: none

Stores a quadlet *quad* to *addr* without requiring alignment.

unaligned-l@

stack: (addr -- quad)
code: none

Fetches a quadlet *quad* from *addr* without requiring alignment.

unaligned-w!

stack: (w addr --)
code: none

Stores a doublet *w* to *addr* without requiring alignment.

unaligned-w@

stack: (addr -- w)
code: none

Fetches doublet *w* from *addr* without requiring alignment.

unloop

stack: (--) (R: loop-sys --)
code: 89

Used within `do` or `?do` loops to discard loop control parameters prior to calling `exit`.

For example:

```
: find-value  ( test-value start-addr -- )
   \ Searches up to 100 locations looking for a test value.
   100 bounds do      ( test-value )
      i c@ over = if  ( test-value )
         ." Found at " i . cr  drop unloop exit
      then
   loop               ( test-value )
   . ." not found" cr
;
```

See also: `exit`, `leave`

unmap

stack: (virt len --)
code: none

Invalidates any existing address translation for the region of virtual address space beginning at *virt* and continuing for *len* bytes. `unmap` does not free either the virtual address space (as with the `release` standard method) or any physical memory that was associated with *virt*.

If the operation fails for any reason, `unmap` uses `throw` to signal the error.

unselect-dev

stack: (--)
code: none

A User Interface extension provided by some implementations (e.g. FirmWorks/Sun).

Destroys the instance chain whose *ihandle* is stored in `my-self`, clears `my-self` and deactivates the active package leaving no active package. Used to reverse the effect of `select`, `select-dev`, `begin-select` or `begin-select-dev`.

See also: "Using select-dev" on page 35.

until

stack: (C: dest-sys --)
(done? --)
generates: `b?branch -offset`

Marks the end of a `begin`...`until` conditional loop. When `until` is encountered, *done?* is removed and tested. If *done?* is *true*, the loop is terminated and execution continues just after the `until`. If *done?* is *false*, execution jumps back to just after the corresponding `begin`.

For example:

```
: probe-loop ( addr -- )
   \ generate a tight 'scope loop until a key is pressed.
   begin dup l@ drop key? until drop
;
```

upc

stack: (char1 -- char2)
code: 81

char2 is the upper case version of *char1*. If *char1* is not a lower case letter, it is left unchanged.

For example:

```
: continue? ( -- continue? )
   ." Want to Continue? Enter Y/N" key dup emit
   upc ascii Y =
;
```

See also: `lcc`

u.r "you dot are"

stack: (u size --)
code: 9C

u is converted according to the value of `base` and then displayed as an unsigned number right-aligned in a field *size* digits wide. A trailing space is *not* displayed.

If the number of digits required to display *u* is greater than *size*, all the digits are displayed with no leading spaces in a field as wide as necessary.

For example:

```
: formatted-output ( -- )
   my-base    h# 8 u.r  ."  base" cr
   my-offset  h# 8 u.r  ."  offset" cr
;
```

use-nvramrc?

stack: (-- enabled?)

This configuration variable is a boolean specifying whether the NVRAM script should be evaluated at system start-up. If `use-nvramrc?` is *true*, the script is evaluated. If `use-nvramrc?` is *false*, the script is not evaluated.

The suggested default value of `use-nvramrc?` is `false`.

user-abort

stack: (… --) (R: … --)
code: 219

Used within an `alarm` routine to signify that the user has typed an abort sequence. When `alarm` finishes, instead of returning to the program that was interrupted by the execution of `alarm`, it enters the Open Firmware command interpreter by calling `abort`.

For example:

```
: test-dev-status  ( -- error? )  … ;
: my-checker  ( -- )  test-dev-status  if  user-abort  then  ;
: install-abort  ( -- )  ['] my-checker d# 10 alarm  ;
```

value

stack: (E: -- x) (x "new-name< >" --)
generates: `new-token|named-token|external-token b(value)`

Creates and initializes a `value` with the name *new-name*. When later executed, *new-name* leaves its value on the stack. The value of *new-name* can be changed with `to`.

For example:

```
ok 123 value foo  foo .
123
ok 456 to foo  foo .
456
```

Open Firmware uses `value` items widely. We encourage the use of `value` instead of `variable`. A `value` acts identically to a `constant` in that it leaves its value on the stack when executed; a `variable` must be fetched to obtained its value. But, unlike a `constant`, the contents of a `value` can be changed. By consistently using `value` (as opposed to intermixing `value` and `variable`), your code will be cleaner and you will not have to wonder whether a given datum should be stored with `!` or `to`, or whether or not you need to use `@` .

In FCode Source, `value` cannot appear inside a colon definition.

value>sym “value to sym”

stack: (n1 -- n1 false | n2 addr len true)
code: none

Defer word to resolve symbol values.

This `defer` word is executed when the symbolic debugger needs to translate a symbol value into its corresponding name. If `value>sym` is present, the disassembler attempts to perform such a translation to display the symbolic representations of the addresses that it displays. The translation is also attempted by `.adr`.

If the symbol table contains a symbol whose value is sufficiently close to, but not greater than, the value *n1*, `value>sym` returns the string *addr len* representing the name of that symbol, the non-negative difference *n2* between the symbol value and *n1*, and *true*. Otherwise, `value>sym` returns *n1* and *false*.

The default action for `value>sym`, when no symbol table is present, is to return *n1* and *false*. A program can provide a symbol table and install a command into `value>sym` with `to` to search that symbol table.

See also: `sym>value`

variable

stack: (E: -- a-addr) ("new-name< >" --)
generates: `new-token|named-token|external-token b(variable)`

Creates an uninitialized `variable` named *new-name*. When later executed, *new-name* leaves its address on the stack. The alignment of the returned address is system-dependent. The address holds a 32-bit value.

The value of *new-name* can be changed with `!` and fetched with `@` .

For example:

```
ok variable foo  123 foo !  foo @ .
123
ok 456 foo ! foo ?
456
```

FirmWorks encourages the use of `value` instead of `variable`. A `value` acts identically to a `constant` in that it leaves its value on the stack when executed; a `variable` must be fetched to obtained its value. But, unlike a `constant`, the contents of a `value` can be changed. By consistently using `value` (as opposed to intermixing `value` and `variable`), your code will be cleaner and you will not have to wonder whether a given datum should be stored with `!` or `to`, or whether or not you need to use `@` .

In FCode Source, `value` cannot appear inside a colon definition.

version1

stack: (--)
code: FD

`version1` may only be used as the first byte of an FCode Program. `version1`:

- Sets the `spread` value to 0 causing the FCode Evaluator to read successive bytes of the current FCode Program from the same address.
- Establishes the use of 8-bit branches.
- Reads an FCode header from the current FCode Program and either discards it or uses it to verify the integrity of the current FCode program in an implementation-dependent manner.

For portability, the preferred way to begin an FCode program in source form is with the `fcode-version1` tokenizer macro. This macro causes the tokenizer to begin the FCode binary with the `version1` byte and an FCode header.

See also: `fcode-version2`, `start0`, `start1`, `start2`, `start4`

w! "double you store"

stack: (w waddr --)
code: 74

The low-order 16-bits of *w* are stored at location *waddr. waddr* must be aligned on a 16-bit boundary.

See also: `rw!`

w, “double you comma”

stack: (w --)
code: D1

Compile 16-bits into the dictionary. The dictionary pointer must be 16-bit-aligned.

See also: `c,`

w@ “double you fetch”

stack: (waddr -- w)
code: 6F

Fetch the 16-bit number stored at *waddr* and leave the result on the stack. *waddr* must be aligned on a 16-bit boundary.

See also: rw@

/w “per double you”

stack: (-- n)
code: 5B

n is the number of address units to a 16-bit word, typically 2.

/w* “per double you star”

stack: (nu1 -- nu2)
code: 67

nu2 is the result of multiplying *nu1* by `/w`. This is the portable way to convert an index into a byte offset.

<w@

stack: (waddr -- n)
code: 70

Fetches the 16-bit number stored at *waddr* and extends its sign into the upper bytes. *waddr* must be 16-bit-aligned.

For example:

```
ok 9123 8000 w!  8000 <w@ .h
ffff9123
ok 8000 w@ .h
9123
```

wa+

stack: (addr1 index -- addr2)
code: 5F

Increments *addr1* by *index* times the value of `/w`. This is the portable way to increment an address.

wa1+

stack: (addr1 -- addr2)
code: 63

Increments *addr1* by the value of `/w`. This is the portable way to increment an address.

wbflip

stack: (w1 -- w2)
code: 80

w2 is the result of exchanging the two low-order bytes of the number *w1*. The two upper bytes of *w1* must be zero, or erroneous results will occur.

wbflips

stack: (waddr len --)
code: 236

Swaps the order of the bytes within each 16-bit word in the memory buffer *waddr len*.

waddr must be 16-bit-aligned. *len* must be a multiple `/w`.

wbsplit

stack: (w -- b1.lo b2.hi)
code: AF

Splits the lower 16 bits of *w* into two separate bytes. All but the least significant 8 bits of each output stack result are zero.

while

stack: (C: dest-sys -- orig-sys dest-sys)
(continue? --)
generates: `b?branch +offset`

Tests the exit condition for a `begin`...`while`...`repeat` conditional loop. When the `while` is encountered, *continue?* is removed from the stack and tested. If *continue?* is *true*, execution continues from just after the `while` through to the `repeat` which then jumps back to just after the `begin`. If *continue?* is *false*, the loop is exited by causing execution to jump ahead to just after the `repeat`.

For example:

```
: probe-loop ( addr -- )
   \ generate a tight 'scope loop until a key is pressed.
   begin key? 0=  while  dup l@ drop  repeat  drop
;
```

"width"

This standard property is associated with `display` devices. The property value is an integer (encoded with `encode-int`) that specifies the number of displayable pixles in the “x” dimension.

window-left

stack: (-- border-width)
code: 166

A `value`, containing the offset (in pixels) of the left edge of the active text area from the left edge of the visible display. The “active text area” is where characters are actually printed. (There is generally a border of unused blank area surrounding it on all sides.) `window-left` contains the size of the left portion of the unused border.

The size of the right portion of the unused border is determined by the difference between `screen-width` and the sum of `window-left` plus the width of the active text area (`#columns` times `char-width`).

This word is initially set to 0, but should always be set explicitly to an appropriate value if you wish to use *any* `fb1-` or `fb8-` utility routines. This can be done with `to`, or it can be set automatically by calling `fb1-install` or `fb8-install`.

When set with `fbx-install`, a calculation is done to set `window-left` so that the available unused border area is evenly split between the left border and the right border. (The calculated value for `window-left` is rounded down to the nearest multiple of 32, though. This allows all pixel-drawing to proceed more efficiently.) If you wish to use `fbx-install` but desire a different value for `window-left`, simply change it with `to` *after* calling `fbx-install`.

window-top

stack: (-- border-height)
code: 165

A `value`, containing the offset (in pixels) of the top of the active text area from the top of the visible display. The “active text area” is where characters are actually printed. (There is generally a border of unused blank area surrounding it on all sides.) `window-top` contains the size of the top portion of the unused border.

The size of the bottom portion of the unused border is determined by the difference between `screen-height` and the sum of `window-top` plus the height of the active text area (`#lines` times `char-height`).

This word is initially set to 0, but should always be set explicitly to an appropriate value if you wish to use *any* `fb1-` or `fb8-` utility routines. This can be done with `to`, or it can be set automatically by calling `fb1-install` or `fb8-install`. When set with `fbx-install`, a calculation is done to set `window-top` so that the available unused border area is evenly split between the top border and the bottom border. If you wish to use `fbx-install` but desire a different value for `window-top`, simply change it with `to` *after* calling `fb`*x*`-install`.

within

stack: (n min max -- min<=n<max?)
code: 45

min<=n<max? is *true* if *n* is between *min* and *max*, inclusive of *min* and exclusive of *max*.

See also: `between`

wljoin

stack: (w.lo w.hi -- quad)
code: 7D

Combines the least significant 16-bits of each of the two input stack arguments into one 32-bit output stack result. All but the least significant 16-bits of *w.lo* and *w.hi* must be zero.

word

stack: (delim "<delims>text<delim>" -- pstr)
code: none

Parses text from the input buffer delimited by *delim*.

words

stack: (--)
code: none

If there is an active package, displays the names of its methods. Otherwise, displays an implementation-dependent subset (preferably the entire set) of the globally-visible Forth commands. In either case, the order of display is to display more-recently-defined names before less-recently-defined names.

The particular words displayed by `words` can be affected by the tokenizer directives `external`, `headers` and `headerless`, and by the state of the configuration variable `fcode-debug?`.

If the FCode program was interpreted from text source, the tokenizer directives have no affect on the words that are displayed.

However, if the FCode program is first tokenized and then evaluated, `words` displays:

- All words which were created by the FCode evaluator while the tokenizer directive `external` was in effect.
- All words created by the FCode evaluator while the tokenizer directive `headers` was in effect *if* the configuration variable `fcode-debug?` was true when the FCode was evaluated.

`words` never displays words created by the FCode evaluator while the `headerless` tokenizer directive was in effect.

wpeek

stack: (waddr -- false | w true)
code: 221

Tries to read the 16-bit word at address *waddr*. Returns *w* and *true* if the access was successful. A *false* return indicates that a read access error occurred. *waddr* must be 16-bit aligned.

wpoke

stack: (w waddr -- okay?)
code: 224

Tries to write the 16-bit word at address *waddr*. Returns *true* if the access was successful. A *false* return indicates that a write access error occurred. *waddr* must be 16-

bit aligned.

Note: `wpoke` may be unreliable on bus adapters that buffer write accesses.

write

stack: (addr len -- actual)
code: none

Writes *len* bytes to the device from the memory buffer beginning at *addr*. Returns *actual*, the number of bytes actually written. If *actual* is less than *len*, the write did not succeed.

Devices of type `network` place additional requirements on their `write` methods:

- network

  The `write` method transmits the network packet of *len* bytes from memory at *addr*, returning the number of bytes actually transmitted. The caller must supply the complete packet including the MAC header with source and destination address.

For some devices, the `seek` method sets the position for the next `write`.

write-blocks

stack: (addr block# #blocks -- #written)
code: none

Writes *#blocks* records of length `block-size` bytes from memory (starting at *addr*) to the device (starting at device block *block#*). Returns *#written*, the number of blocks actually written.

If the device is not capable of random access (e.g. a sequential access tape device), *block#* is ignored.

wxjoin

stack: (w.lo w.2 w.3 w.hi -- o)
code: 244

Join four doublets to form an octlet. The high-order bits of each of the doublets are ignored.

This function is only available on 64-bit implementations.

x, “ecks comma”

stack: (o --)
code: 245

Compile an octlet, o, into the dictionary (doublet-aligned).

This function is only available on 64-bit implementations.

x@ “ecks fetch”

stack: (oaddr -- o)
code: 246

Fetch octlet from an octlet aligned address.

This function is only available on 64-bit implementations.

x! “ecks store”

stack: (o oaddr --)
code: 247

Store octlet to an octlet aligned address.

This function is only available on 64-bit implementations.

/x “per ecks”

stack: (-- n)
code: 248

Number of address units in an octlet, typically eight.

This function is only available on 64-bit implementations.

/x* “per ecks star”

stack: (nu1 -- nu2)
code: 249

Multiply nu1 by the value of /x.

This function is only available on 64-bit implementations.

xa+ “ecks ay plus”

stack: (addr1 index -- addr2)
code: 24A

Increment addr1 by index times the value of /x.

This function is only available on 64-bit implementations.

xa1+ “ecks ay one plus”

stack: (addr1 -- addr2)
code: 24B

Increment addr1 by the value of /x.

This function is only available on 64-bit implementations.

xbflip

stack: (oct1 -- oct2)
code: 24C

Reverse the bytes within an octlet.

This function is only available on 64-bit implementations.

xbflips

stack: (oaddr len --)
code: 24D

Reverse the bytes within each octlet in the given region. The region begins at oaddr and spans len bytes. The behavior is undefined if len is not a multiple of /x.

This function is only available on 64-bit implementations.

xbsplit

stack: (o -- b.lo b.2 b.3 b.4 b.5 b.6 b.7 b.hi)
code: 24E

Split an octlet into 8 bytes. The high-order bits of each of the eight bytes are zero.

This function is only available on 64-bit implementations.

xlflip

stack: (oct1 -- oct2)
code: 24F

Reverse the quadlets within an octlet. The bytes within each quadlet are not reversed.

This function is only available on 64-bit implementations.

xlflips

stack: (oaddr len --)
code: 250

Reverse the quadlets within each octlet in the given region. The bytes within each quadlet are not reversed. The region begins at oaddr and spans len bytes. The behavior is undefined if len is not a multiple of /x.

This function is only available on 64-bit implementations.

xlsplit

stack: (o -- quad.lo quad.hi)
code: 251

Split on octlet into 2 quadlets. The high-order bits of each of the two quadlets are zero.

This function is only available on 64-bit implementations.

xor

stack: (x1 x2 -- x3)
code: 25

x3 is the bit-by-bit exclusive-or of *x1* with *x2*.

xwflip

stack:	(oct1 -- oct2)
code:	252

Reverse the doublets within an octlet. The bytes within each doublet are not reversed.

This function is only available on 64-bit implementations.

xwflips

stack:	(oaddr len --)
code:	253

Reverse the doublets within each octlet in the given region. The bytes within each doublet are not reversed. The region begins at oaddr and spans len bytes. The behavior is undefined if len is not a multiple of /x.

This function is only available on 64-bit implementations.

xwsplit

stack:	(o -- w.lo w.2 w.3 w.hi)
code:	254

Split an octlet into 4 doublets. The high-order bits of each doublet are zero.

This function is only available on 64-bit implementations.

A

FCode Reference

FCode Primitives

This appendix contains three lists:

- FCodes sorted according to functional group
- FCodes sorted by byte value
- FCodes sorted alphabetically by name

Each of these lists consist of one or more tables. The tables describe FCodes currently supported by Open Firmware. Both the FCode token values and Forth names are included. A token value entry of `TG` indicates a tokenizer-generated sequence, while `-` indicates a tokenizer directive.

FCodes by Function

Table 40 Stack Manipulation

Value	Function	Stack	Description
51	depth	(-- u)	How many items on stack?
46	drop	(x --)	Removes top item from the stack
52	2drop	(x1 x2 --)	Removes 2 items from the stack
TG	3drop	(x1 x2 x3 --)	Removes 3 items from the stack
47	dup	(x -- x x)	Duplicates the top stack item
53	2dup	(x1 x2 -- x1 x2 x1 x2)	Duplicates 2 stack items
TG	3dup	(x1 x2 x3 -- x1 x2 x3 x1 x2 x3)	Copies top 3 stack items
50	?dup	(x -- 0 | x x)	Duplicates the top stack item if it is non-zero
4D	nip	(x1 x2 -- x2)	Discards the second stack item
48	over	(x1 x2 -- x1 x2 x1)	Copies the second stack item to the top of stack
54	2over	(x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2)	Copies 2 stack items
4E	pick	(xu … x1 x0 u -- xu … x1 x0 xu)	Copies u-th stack item
30*	>r	(x --) (R: -- x)	Moves a stack item to the return stack*
31*	r>	(-- x) (R: x --)	Moves item from return stack to data stack*
32	r@	(-- x) (R: x -- x)	Copies the top of the return stack to the data stack
4F	roll	(xu … x1 x0 u -- xu-1 … x1 x0 xu)	Rotates u stack items
4A	rot	(x1 x2 x3 -- x2 x3 x1)	Rotates 3 stack items
4B	-rot	(x1 x2 x3 -- x3 x1 x2)	Shuffles top 3 stack items
56	2rot	(x1 x2 x3 x4 x5 x6 -- x3 x4 x5 x6 x1 x2)	Rotates 3 pairs of stack items
49	swap	(x1 x2 -- x2 x1)	Exchanges the top 2 stack items
55	2swap	(x1 x2 x3 x4 -- x3 x4 x1 x2)	Exchanges 2 pairs of stack items
4C	tuck	(x1 x2 -- x2 x1 x2)	Copies the top stack item belowthe second item
* USE THIS FCODE CAUTIOUSLY.			

Table 41 Single-Precision Arithmetic Operations

Value	Function	Stack	Description
1E	+	(nu1 nu2 -- sum)	Adds *nu1* + *nu2*.
1F	-	(nu1 nu2 -- diff)	Subtracts *nu1* - *nu2*.
20	*	(nu1 nu2 -- prod)	Multiplies nu1 times nu2
21	/	(n1 n2 -- quot)	Divide *n1* by *n2*; remainder is discarded.
TG	1+	(nu1 -- nu2)	Adds one.
TG	1-	(nu1 -- nu2)	Subtracts one.
TG	2+	(nu1 -- nu2)	Adds two.
TG	2-	(nu1 -- nu2)	Subtracts two.
2D	abs	(n -- u)	Absolute value.
AC	bounds	(start len -- len+start start)	Converts *start,len* to *end,start* for do or ?do loop.
2F	max	(n1 n2 -- n1|n2)	n3 is maximum of *n1* and *n2*
2E	min	(n1 n2 -- n1|n2)	n3 is minimum of *n1* and *n2*

Table 41 Single-Precision Arithmetic Operations *(Continued)*

Value	Function	Stack	Description
22	mod	(n1 n2 -- rem)	Remainder of *n1 / n2*.
TG	*/mod	(n1 n2 n3 -- rem quot)	Remainder, quotient of *n1 * n2 / n3*.
2A	/mod	(n1 n2 -- rem quot)	Remainder, quotient of n1/n2
2C	negate	(n1 -- n2)	Changes the sign of *n1*
2B	u/mod	(u1 u2 -- urem uquot)	Divide unsigned one-cell number by an unsigned one-cell number; yield one-cell remainder and quotient.

Table 42 Bitwise Logical Operations

Value	Function	Stack	Description
TG	<<	(x1 u -- x2)	Synonym for `lshift`.
TG	>>	(x1 u -- x2)	Synonym for `rshift`.
59	2*	(x1 -- x2)	Multiplies by 2
57	2/	(x1 -- x2)	Divides by 2
29	>>a	(x1 u -- x2)	Arithmetic right shifts n1 by u places
23	and	(x1 x2 -- x3)	Bitwise logical AND
26	invert	(x1 -- x2)	Invert all bits of *x1*
27	lshift	(x1 u -- x2)	Left-shift *x1* by *u* bits. Zero-fill low bits.
TG	not	(x1 -- x2)	Synonym for invert
24	or	(x1 x2 -- x3)	Bitwise logical OR
28	rshift	(x1 u -- x2)	Right-shift *x1* by *u* bits. Zero-fill high bits.
58	u2/	(x1 -- x2)	Logical right shift 1 bit; zero shifted into high bit.
25	xor	(x1 x2 -- x3)	Bitwise exclusive OR.

Table 43 Double Number Arithmetic Operations

Value	Function	Stack	Description
D8	d+	(d1 d2 --d.sum)	Adds two double numbers yielding a double number
D9	d-	(d1 d2 -- d.diff)	Subtracts two double numbers yielding a double number
D4	um*	(u1 u2 -- ud.prod)	Multiplies two unsigned numbers yielding an unsigned double number product.
D5	um/mod	(ud u -- urem uquot)	Divides an unsigned double number by an unsigned yielding an unsigned remainder and quotient

Table 44 Memory Access

Value	Function	Stack	Description
72	!	(x a-addr --)	Stores a number into the variable at a-addr
6C	+!	(nu a-addr --)	Adds nu to the number stored in the variable at a-addr
77	2!	(x1 x2 a-addr --)	Stores 2 numbers at a-addr; x2 at lower address

Table 44 Memory Access *(Continued)*

Value	Function	Stack	Description
76	2@	(a-addr -- x1 x2)	Fetches 2 numbers from a-addr; x2 from lower address
6D	@	(a-addr -- x)	Fetches a number from the variable at a-addr
TG	?	(a-addr --)	Displays the number at a-addr
75	c!	(byte addr --)	Stores low byte of n at addr
71	c@	(addr -- byte)	Fetches a byte from addr
TG	blank	(addr len --)	Sets len bytes of memory to ASCII space, starting at addr
7A	comp	(addr1 addr2 len -- n)	Compares two byte arrays including case. n=0 if same
TG	erase	(addr len --)	Sets len bytes of memory to zero, starting at addr
79	fill	(addr len byte --)	Sets len bytes of memory to value byte
0228	lbflips	(qaddr len --)	Reverses bytes within each quadlet in given region
0237	lwflips	(qaddr len --)	Exchanges doublets within quadlets in qaddr len
73	l!	(quad qaddr --)	Stores the quadlet at qaddr, must be 32-bit aligned
6E	l@	(qaddr -- quad)	Fetches the quadlet at qaddr, must be 32-bit aligned
78	move	(src-addr dest-addr len --)	Copies len bytes from src-addr to dest-addr, handles overlap correctly.
6B	off	(a-addr --)	Stores false (32-bit 0) at a-addr
6A	on	(a-addr --)	Stores true (32-bit -1) at a-addr
0236	wbflips	(waddr len --)	Exchanges bytes within doublets in the specified region
74	w!	(w waddr --)	Stores a doublet at waddr, must be 16-bit aligned
6F	w@	(waddr -- w)	Fetches the unsigned doublet at waddr, must be 16-bit aligned
70	<w@	(waddr -- n)	Fetches the signed doublet at waddr, must be 16-bit aligned

The memory access commands listed in Table 45 are available only on 64-bit Open Firmware implementations.

Table 45 64-Bit Memory Access

Value	Function	Stack	Description
0242	`<l@`	(qaddr -- n)	Fetch quadlet from qaddr, sign-extended.
0246	`x@`	(oaddr -- o)	Fetch octlet from an octlet aligned address.
0247	`x!`	(o oaddr --)	Store octlet to an octlet aligned address.
024D	xbflips	(oaddr len --)	Reverse the bytes within each octlet in the given region.The behavior is undefined if *len* is not a multiple of `/x`.
0250	xlflips	(oaddr len --)	Reverse the quadlets within each octlet in the given region. The bytes within each quadlet are not reversed. The behavior is undefined if *len* is not a multiple of `/x`.
0253	xwflips	(oaddr len --)	Reverse the doublets within each octlet in the given region. The bytes within each doublet are not reversed. The behavior is undefined if *len* is not a multiple of `/x`.

Table 46 Atomic Access

Value	Function	Stack	Description
0230	rb@	(addr -- byte)	Reads the 8-bit value at the given address, atomically
0231	rb!	(byte addr --)	Writes the 8-bit value at the given address, atomically
0232	rw@	(waddr -- w)	Reads the doublet at the given address, atomically
0233	rw!	(w waddr --)	Writes the doublet at the given address, atomically
0234	rl@	(qaddr -- quad)	Reads the quadlet at the given address, atomically
0235	rl!	(quad qaddr --)	Writes the quadlet at the given address, atomically

The atomic access commands listed in Table 47 are available only on 64-bit Open Firmware implementations.

Table 47 64-Bit Atomic Access

Value	Function	Stack	Description
022E	rx@	(oaddr -- o)	Reads the octlet at the given address, atomically
022F	rx!	(o oaddr --)	Writes the octlet at the given address, atomically

Table 48 Data Exception Tests

Value	Function	Stack	Description
0220	cpeek	(addr -- false | byte true)	Reads 8-bit value at given address, returns false if unsuccessful
0221	wpeek	(waddr -- false | w true)	Reads doublet at given address, returns false if unsuccessful
0222	lpeek	(qaddr -- false | quad true)	Reads quadlet at given address, returns false if unsuccessful
0223	cpoke	(byte addr -- okay?)	Writes 8-bit value at given address, returns false if unsuccessful
0224	wpoke	(w waddr -- okay?)	Writes doublet at given address, returns false if unsuccessful
0225	lpoke	(quad qaddr -- okay?)	Writes quadlet at given address, returns false if unsuccessful

Table 49 Comparison Operations

Value	Function	Stack	Description
36	0<	(n -- less-than-0?)	True if n < 0
37	0<=	(n -- less-or-equal-to-0?)	True if n <= 0
35	0<>	(n -- not-equal-to-0?)	True if n <> 0
34	0=	(nulflag -- equal-to-0?)	True if n = 0, also inverts any flag
38	0>	(n -- greater-than-0?)	True if n > 0
39	0>=	(n -- greater-or-equal-to-0?)	True if n >= 0
3A	<	(n1 n2 -- less?)	True if n1 < n2
43	<=	(n1 n2 -- less-or-equal?)	True if n1 <= n2
3D	<>	(x1 x2 -- not-equal?)	True if x1 <> x2
3C	=	(x1 x2 -- equal?)	True if x1 = x2
3B	>	(n1 n2 -- greater?)	True if n1 > n2

Table 49 Comparison Operations *(Continued)*

Value	Function	Stack	Description
42	>=	(n1 n2 -- greater-or-equal?)	True if n1 >= n2
44	between	(n min max -- min<=n<=max?)	True if min <= n <= max
TG	false	(-- false)	The value false
TG	true	(-- true)	The value true
40	u<	(u1 u2 -- unsigned-less?)	True if u1 < u2, unsigned
3F	u<=	(u1 u2 -- unsigned-less-or-equal?)	True if u1 <= u2, unsigned
3E	u>	(u1 u2 -- unsigned-greater?)	True if u1 > u2, unsigned
41	u>=	(u1 u2 -- unsigned-greater-or-equal?)	True if u1 >= u2, unsigned
45	within	(n min max -- min<=n<max?)	True if min <= n < max

Table 50 Text Input

Value	Function	Stack	Description
-	(	([text<)> --)	Begins a comment (ignored)
-	\	(--)	Ignore rest of line (comment)
TG	ascii	([text< >] -- char)	ASCII value of next character
TG	control	([text< >] -- char)	Interprets next character as ASCII control character
8E	key	(-- char)	Reads a character from the keyboard
8D	key?	(-- pressed?)	True if a key has been typed on the keyboard
TG	accept	(addr len1 -- len2)	Gets an edited input line, stores it at addr
8A	expect	(addr len --)	Gets a line of edited input from the keyboard; stores it at addr
88	span	(-- a-addr)	Variable containing the number of characters read by expect

Table 51 ASCII Constants

Value	Function	Stack	Description
AB	bell	(-- 0x07)	The ASCII code for the bell character; decimal 7
A9	bl	(-- 0x20)	The ASCII code for the space character; decimal 32
AA	bs	(-- 0x08)	The ASCII code for the backspace character; decimal 8
TG	carret	(-- 0x0D)	The ASCII code for the carriage return character; decimal 13
TG	linefeed	(-- 0x0A)	The ASCII code for the linefeed character; decimal 10
TG	newline	(-- n)	The ASCII code for the newline character; decimal 10

Table 52 Numeric Input

Value	Function	Stack	Description
A4	-1	(-- -1)	Constant -1
A5	0	(-- 0)	Constant 0
A6	1	(-- 1)	Constant 1
A7	2	(-- 2)	Constant 2

Table 52 Numeric Input *(Continued)*

Value	Function	Stack	Description
A8	3	(-- 3)	Constant 3
TG	d#	([number< >] -- n)	Interprets next number in decimal
-	decimal	(--)	If outside definition, change numeric conversion base to decimal
TG	h#	([number< >] -- n)	Interprets next number in hexadecimal
-	hex	(--)	If outside definition, change numeric conversion base to hexadecimal
TG	o#	([number< >] -- n)	Interprets next number in octal
-	octal	(--)	If outside definition, change numeric conversion base to octal

Table 53 Numeric Primitives

Value	Function	Stack	Description
99	u#	(u1 -- u2)	Converts a digit in pictured numeric output
97	u#>	(u -- str len)	Ends pictured numeric output
96	<#	(--)	Initializes pictured numeric output
C7	#	(ud1 -- ud2)	Converts a digit in pictured numeric output conversion
C9	#>	(ud -- str len)	Ends pictured numeric output conversion
A0	base	(-- a-addr)	Variable containing number base
A3	digit	(char base -- digit true | char false)	Converts a character to a digit
95	hold	(char --)	Inserts the char in the pictured numeric output string
C8	#s	(ud -- 0 0)	Converts remaining digits in pictured numeric output
9A	u#s	(u1 -- u2)	Converts rest of the digits in pictured numeric output
98	sign	(n --)	Sets sign of pictured output
A2	$number	(addr len -- true | n false)	Converts a string to a number

Table 54 Numeric Output

Value	Function	Stack	Description
9D	.	(nu --)	Displays a number
TG	.d	(n --)	Displays number in decimal
TG	decimal	(--)	If inside definition, output in decimal
TG	.h	(n --)	Displays number in hexadecimal
TG	hex	(--)	If inside definition, output in hexadecimal
TG	octal	(--)	If inside definition, output in octal
9E	.r	(n size --)	Displays a number in a fixed width field
9F	.s	(… -- …)	Displays the contents of the data stack
TG	s.	(n --)	Displays n as a signed number
9B	u.	(u --)	Displays an unsigned number
9C	u.r	(u size --)	Prints an unsigned number in a fixed width field

Table 55 General-purpose Output

Value	Function	Stack	Description
TG	.(	([text<)>] --)	Displays a string now
91	(cr	(--)	Outputs ASCII CR character; decimal 13
92	cr	(--)	Starts a new line of display output
8F	emit	(char --)	Displays the character
TG	space	(--)	Outputs a single space character
TG	spaces	(cnt --)	Outputs cnt spaces
90	type	(text-str text-len --)	Displays n characters

Table 56 Formatted Output

Value	Function	Stack	Description
94	#line	(-- a-addr)	Variable holding the line number on the output device
93	#out	(-- a-addr)	Variable holding the column number on the output device

Table 57 `begin` Loops

Value	Function	Stack	Description
TG	again	(C: dest-sys --)	Ends begin…again (infinite) loop
TG	begin	(C: -- dest-sys) (--)	Starts conditional loop
TG	repeat	(C: orig-sys dest-sys --) (--)	Returns to loop start
TG	until	(C: dest-sys --) (done? --)	If true, exits begin…until loop
TG	while	(C: dest-sys -- orig-sys dest-sys) (continue? --)	If true, continues begin…while…repeat loop, else exits loop

Table 58 Conditionals

Value	Function	Stack	Description
TG	if	(C: -- orig-sys) (do-next? --)	If true, executes next FCode(s)
TG	else	(C: orig-sys1 -- orig-sys2) (--)	(optional) Executes next FCode(s) if if failed
TG	then	(C: orig-sys --) (--)	Terminates if…else…then

Table 59 `case` Statements

Value	Function	Stack	Description
TG	case	(C: -- case-sys) (sel -- sel)	Begins a case (multiple selection) statement

Table 59 case Statements *(Continued)*

Value	**Function**	**Stack**	**Description**
TG	endcase	(C: case-sys --) (sel --)	Marks end of a case statement
TG	of	(C: case-sys1 -- case-sys2 of-sys) (sel of-val -- sel | <nothing>)	Returns to loop start
TG	endof	(C: case-sys1 of-sys -- case-sys2) (--)	If true, exits begin…until loop

Table 60 do Loops

Value	**Function**	**Stack**	**Description**
TG	do	(C: -- dodest-sys) (limit start --) (R: -- sys)	Loops, index *start* to *end-1* inclusive
TG	?do	(C: -- dodest-sys) (limit start --) (R: -- sys)	Like do, but skips loop if *end = start*
19	i	(-- index) (R: sys -- sys)	Returns current loop index value
1A	j	(-- index) (R: sys -- sys)	Returns value of next outer loop index
TG	leave	(--) (R: sys --)	Exits do loop immediately
TG	?leave	(exit? --) (R: sys --)	If flag is true, exits do loop
TG	loop	(C: dodest-sys --) (--) (R: sys1 -- <nothing> | sys2)	Increments index, returns to do
TG	+loop	(C: dodest-sys --) (delta --) (R: sys1 -- <nothing> | sys2)	Increments by n, returns to do. If n<0, index *start* to *end*
89	unloop	(--) (R: sys --)	Discards loop control parameters

Table 61 Control Words

Value	**Function**	**Stack**	**Description**
1D	execute	(… xt -- ???)	Executes the word whose compilation address is on the stack
33	exit	(--) (R: sys --)	Returns from the current word

Table 62 Strings

Value	**Function**	**Stack**	**Description**
TG	"	([text<">< >] -- text-str text-len)	Collects a string
TG	s"	([text<“>] -- test-str text-len)	Gathers the immediately-following string
84	count	(pstr -- str len)	Unpacks a packed string
82	lcc	(char1 -- char2)	Converts char to lower case
83	pack	(str len addr -- pstr)	Makes a packed string from addr len, placing it at pstr
81	upc	(char1 -- char2)	Converts char to upper case
0240	left-parse-string	(str len char -- R-str R-len L-str L-len)	Splits a string at the given delimiter (which is discarded)
011B	parse-2int	(str len -- val.lo val.hi)	Converts a string into a physical address and space

Table 63 Defining Words

Value	Function	Stack	Description
TG	: (colon) *name*	(E: … -- ???) (--)	Begins colon definition
TG	; (semicolon)	(--)	Ends colon definition
-	alias	(E: … -- ???) (“new-name< >old-name< >” --)	Creates newname with behavior of oldname
TG	buffer:	(E: -- a-addr) (len "new-name< >" --)	Creates data array of size bytes
TG	constant	(E: -- x) (x "new-name< >" --)	Creates a constant
TG	create	(E: -- a-addr) ("new-name< >" --)	Generic defining word
TG	defer	(E: … -- ???) ("new-name< >" --)	Execution vector (change with is)
TG	field	(E: addr -- addr+offset) (offset size "new-name< >" -- offset+size)	Creates a named offset pointer
C0	instance	(--)	Declare a data type to be local
TG	struct	(-- 0)	Initializes for field creation
TG	variable	(E: -- a-addr) ("new-name< >"--)	Creates a data variable
TG	value	(E: -- x) (x "new-name< >"--)	Creates named value-type variable (change with is)

Table 64 Dictionary Compilation

Value	Function	Stack	Description
D3	,	(x --)	Places a number in the dictionary
D0	c,	(byte --)	Places a byte in the dictionary
AD	here	(-- addr)	Address of top of dictionary
D2	l,	(quad --)	Places a quadlet in the dictionary
D1	w,	(w --)	Places a doublet in the dictionary
TG	allot	(len --)	Allocates len bytes in the dictionary
TG	to	(param [old-name< >] --)	Changes value in a defer word or a value
DD	compile	(--)	Compiles following command at run time
DC	state	(-- a-addr)	Variable containing true if in compilation state

The dictionary compilation commands listed in Table 65 are available only on 64-bit Open Firmware implementations.

Table 65 64-Bit Dictionary Compilation

Value	Function	Stack	Description
0245	`x,`	(o --)	Compile an octlet, o, into the dictionary (doublet-aligned).

Table 66 Dictionary Search

Value	Function	Stack	Description
TG	'	("old-name< >" -- xt)	Finds the word (while executing)
TG	['] *name*	(-- xt)	Finds word (while compiling)
CB	$find	(name-str name-len -- xt true | name-str name-len false)	Finds a name in the dictionary
CD	eval	(... str len -- ???)	Executes Forth commands within a string
CD	evaluate	(... str len -- ???)	Interprets Forth text from the given string

Table 67 Address Arithmetic

Value	Function	Stack	Description
AE	aligned	(n1 -- n1|a-addr)	Increases *n1* if necessary to yield a variable aligned address.
5A	/c	(-- n)	Address increment for a byte; 1
TG	/c*	(nu1 -- nu2)	Synonym for `chars`
5E	ca+	(addr1 index -- addr2)	Increments addr1 by index times /c
TG	ca1+	(addr1 -- addr2)	Synonym for `chars+`
65	cell+	(addr1 -- addr2)	Increments addr1 by /n
69	cells	(nu1 -- nu2)	Multiplies by /n
62	char+	(addr1 -- addr2)	Increments addr1 by /c
66	chars	(nu1 -- nu2)	Multiplies by /c
5C	/l	(-- n)	Address increment for a quadlet;
68	/l*	(nu1 -- nu2)	Multiplies by /l
60	la+	(addr1 index -- addr2)	Increments addr1 by index times /l
64	la1+	(addr1 -- addr2)	Increments addr1 by /l
5D	/n	(-- n)	Address increment for a normal;
TG	/n*	(nu1 -- nu2)	Synonym for `cells`
61	na+	(addr1 index -- addr2)	Increments addr1 by index times /n
TG	na1+	(addr1 -- addr2)	Synonym for `cell+`
5B	/w	(-- n)	Address increment for a doublet;
67	/w*	(nu1 -- nu2)	Multiplies by /w
5F	wa+	(addr1 index -- addr2)	Increments addr1 by index times /w
63	wa1+	(addr1 -- addr2)	Increments addr1 by /w

The address arithmetic commands listed in Table 68 are available only on 64-bit Open Firmware implementations.

Table 68 64-Bit Address Arithmetic

Value	Function	Stack	Description
0248	`/x`	(-- n)	Number of address units in an octlet, typically eight.

Table 68 64-Bit Address Arithmetic *(Continued)*

Value	Function	Stack	Description
0249	/x*	(nu1 -- nu2)	Multiply *nu1* by the value of /x.
024A	xa+	(addr1 index -- addr2)	Increment *addr1* by *index* times the value of /x.
024B	xa1+	(addr1 -- addr2)	Increment *addr1* by the value of /x.

Table 69 Data Type Conversion

Value	Function	Stack	Description
7F	bljoin	(bl.lo b2 b3 b4.hi -- quad)	Joins four bytes to form a quadlet
B0	bwjoin	(b.lo b.hi -- w)	Joins two bytes to form a doublet
0227	lbflip	(quad1 -- quad2)	Reverses the bytes within a quadlet
7E	lbsplit	(quad -- b.lo b2 b3 b4.hi)	Splits a quadlet into four bytes
7E	lwflip	(quad1 -- quad2)	Swaps the doublets within a quadlet
7C	lwsplit	(quad -- w1.lo w2.hi)	Splits a quadlet into two doublets
80	wbflip	(w1 -- w2)	Swaps the bytes within a doublet
AF	wbsplit	(w -- b1.lo b2.hi)	Splits a doublet into two bytes
7D	wljoin	(w.lo w.hi -- quad)	Joins two doublets to form a quadlet

The data type conversion commands listed in Table 70 are available only on 64-bit Open Firmware implementations.

Table 70 64-Bit Data Type Conversion

Value	Function	Stack	Description
0241	bxjoin	(b.lo b.2 b.3 b.4 b.5 b.6 b.7 b.hi -- o)	Join eight bytes to form an octlet.
0243	lxjoin	(quad.lo quad.hi -- o)	Join two quadlets to form an octlet.
0244	wxjoin	(w.lo w.2 w.3 w.hi -- o)	Join four doublets to form an octlet.
024C	xbflip	(oct1 -- oct2)	Reverse the bytes within an octlet.
024E	xbsplit	(o -- b.lo b.2 b.3 b.4 b.5 b.6 b.7 b.hi)	Split an octlet into eight bytes.
024F	xlflip	(oct1 -- oct2)	Reverse the quadlets within an octlet. The bytes within each quadlet are not reversed.
0251	xlsplit	(o -- quad.lo quad.hi)	Split on octlet into two quadlets.
0252	xwflip	(oct1 -- oct2)	Reverse the doublets within an octlet. The bytes within each doublet are not reversed.
0254	xwsplit	(o -- w.lo w.2 w.3 w.hi)	Split an octlet into four doublets.

Table 71 Memory Buffers Allocation

Value	Function	Stack	Description
8B	alloc-mem	(len -- a-addr)	Allocates nbytes of memory and returns its address
8C	free-mem	(a-addr len --)	Frees memory allocated by alloc-mem

Table 72 Miscellaneous Operators

Value	Function	Stack	Description
86	>body	(xt -- a-addr)	Finds parameter field address from compilation address
85	body>	(a-addr -- xt)	Finds compilation address from parameter field address
DA	get-token	(FCode# -- xt immediate?)	Converts FCode Number to function execution token
DB	set-token	(xt immediate? FCode# --)	Assigns FCode Number to existing function
00	end0	(--)	Marks the end of FCode
FF	end1	(--)	Alternates form for end0 (not recommended)
TG	fcode-version1	(--)	Begins FCode program
023E	byte-load	(addr xt --)	Interprets FCode beginning at location `addr`
-	fload	([filename<cr>] --)	Begins tokenizing *filename*
-	headerless	(--)	Creates new names with new-token (no name fields)
-	headers	(--)	Creates new names with named-token (default)
7B	noop	(--)	Does nothing
CC	offset16	(--)	All further branches use 16-bit offsets (instead of 8-bit)
-	tokenizer[	(--)	Begins tokenizer program commands
-	]tokenizer	(--)	Ends tokenizer program commands
TG	fcode-version2	(--)	Begins 2.0 FCode program, compiles start1
-	external	(--)	Creates new names with external-token

Table 73 Internal Operators, (invalid for program text)

Value	Function	Stack	Description
01-0F			First byte of a two byte FCode
10	b(lit)	(-- n) (F: /FCode-num32/ --)	Followed by 32-bit#. Compiled by numeric data
11	b(')	(-- xt) (F: /FCode#/ --)	Followed by a token (1 or 2-byte code) . Compiled by ['] or ’
12	b(")	(-- str len) (F: /FCode-string/ --)	Followed by count byte, text. Compiled by " or ."
C3	b(to)	(x --)	Compiled by `to`
FD	version1	(--)	Compiled by fcode-version1 as the first FCode byte followed by a reserved byte, the FCode checksum (2 bytes) and the FCode length (4 bytes).
13	bbranch	(--) (F: /FCode-offset/ --)	Followed by offset. Compiled by else or again
14	b?branch	(don’t-branch? --) (F: /FCode-offset/ --)	Followed by offset. Compiled by if or until
15	b(loop)	(--) (F: /FCode-offset/ --)	Followed by offset. Compiled by loop
16	b(+loop)	(delta --) (F: /FCode-offset/ --)	Followed by offset. Compiled by +loop
17	b(do)	(limit start --) (F: /FCode-offset/ --)	Followed by offset. Compiled by do

Table 73 Internal Operators, (invalid for program text) *(Continued)*

Value	Function	Stack	Description
18	b(?do)	(limit start --) (F: /FCode-offset/ --)	Followed by offset. Compiled by ?do
1B	b(leave)	(F: --)	Compiled by leave or ?leave
B1	b(<mark)	(F: --)	Compiled by begin
B2	b(>resolve)	(--) (F: --)	Compiled by else or then
C4	b(case)	(sel -- sel) (F: --)	Compiled by case
C5	b(endcase)	(sel --) (F: --)	Compiled by endcase
C6	b(endof)	(--) (F: /FCode-offset/ --)	Compiled by endof
1C	b(of)	(sel of-val -- sel | <nothing>) (F: /FCode-offset/ --)	Followed by offset. Compiled by of
B5	new-token	(--) (F: /FCode#/ --)	Followed by table#, code#, token-type. Compiled by any defining word. Headerless, not used normally.
B6	named-token	(--) (F: /FCode-string FCode#/ --)	Followed by packed string (count,text), table#, code#, token-type. Compiled by any defining word (: value constant etc.)
B7	b(:)	(E: ... -- ???) (F: -- colon-sys)	Token-type compiled by :
B8	b(value)	(E: -- x) (F: x --)	Token-type compiled by value
B9	b(variable)	(E: -- a-addr) (F: --)	Token-type compiled by variable
BA	b(constant)	(E: -- n) (F: n --)	Token-type compiled by constant
BB	b(create)	(E: -- a-addr) (F: --)	Token-type compiled by create
BC	b(defer)	(E: ... -- ???) (F: --)	Token-type compiled by defer
BD	b(buffer:)	(E: -- a-addr) (F: size --)	Token-type compiled by buffer:
BE	b(field)	(E: addr -- addr+offset) (F: offset size -- offset+size)	Token-type compiled by field
C2	b(;)	(--) (F: colon-sys --)	End a colon definition. Compiled by ;
CA	external-token	(--) (F: /FCode-string FCode#/ --)	vt
F0	start0	(--)	Like start1, but fetches successive tokens from same address
F1	start1	(--)	Compiled by fcode-version2 as the first FCode byte followed by a reserved byte, the FCode checksum (2 bytes) and the FCode length (4 bytes). Uses 16-bit branches. Fetches successive tokens from consecutive addresses
F2	start2	(--)	Like start1, but fetches successive tokens from consecutive 16-bit addresses
F3	start4	(--)	Like start1, but fetches successive tokens from consecutive 32-bit addresses

Table 74 Memory Allocation

Value	Function	Stack	Description
0105	free-virtual	(virt size --)	Frees virtual memory from memmap, dma-alloc,or map-low

Table 75 Properties

Value	Function	Stack	Description
0110	property	(prop-addr prop-len name-str name-len --)	Declares a property with the given value structure, for the given name string.
021E	delete-property	(nam-str nam-len --)	Deletes the property with the given name
0115	encode-bytes	(data-addr data-len -- prop-addr prop-len)	Converts a byte array into an prop-format string
0111	encode-int	(n -- prop-addr prop-len)	Converts a number into an prop-format string
0113	encode-phys	(phys.lo … phys.hi -- prop-addr prop-len)	Converts physical address and space into an prop-format string
0114	encode-string	(str len -- prop-addr prop-len)	Converts a string into an prop-format string
0112	encode+	(prop-addr1 prop-len1 prop-addr2 prop-len2 -- prop-addr3 prop-len3)	Merges two prop-format strings. They must have been created sequentially
TG	decode-bytes	(prop-addr1 prop-len1 data-len -- prop-addr2 prop-len2 data-addr data-len)	Decodes a byte array from a `prop-encoded-array`
021B	decode-int	(prop-addr1 prop-len1 -- prop-addr2 prop-len2 n)	Converts the beginning of an prop-format string to an integer
021C	decode-string	(prop-addr1 prop-len1 -- prop-addr2 prop-len2 str len)	Converts the beginning of an prop-format string to a normal string
0128	decode-phys	(prop-addr1 prop-len1 -- prop-addr2 prop-len2 phys.lo … phys.hi)	Decode a unit-address from a `prop-encoded-array`
021A	get-my-property	(nam-str nam-len -- true | prop-addr prop-len false)	Returns the prop-format string for the given property name
021D	get-inherited-property	(nam-str nam-len -- true | prop-addr prop-len false)	Returns the value string for the given property, searches parents’ properties if not found
021F	get-package-property	(name-str name-len phandle -- true | prop-addr prop-len false)	Returns the prop-format string for the given property name in the package "phandle"

Table 76 Commmonly-used Properties

Value	Function	Stack	Description
0116	reg	(phys.lo … phys.hi size --)	Declares location and size of device registers
0119	model	(str len --)	Declares model number for this device, such as " SUNW,501-1415-01"
011A	device-type	(str len --)	Declares type of device, e.g. " display", " block", " byte", " network", or " serial"
TG	name	(addr len --)	Declares SunOS driver name, as in " SUNW,zebra"
0201	device-name	(str len --)	Creates the "name" property with the given value

Table 77 System Version Information

Value	Function	Stack	Description
87	fcode-revision	(-- n)	Returns major/minor FCode interface version

Table 78 Device Node Creation

Value	Function	Stack	Description
011F	new-device	(--)	Creates a new device node as a child of the active package. and makes the new node the active package.
0127	finish-device	(--)	Completes a device node that was created by `new-device`.

Table 79 Self-test Utility Routines

Value	Function	Stack	Description
0120	diagnostic-mode?	(-- diag?)	Returns "true" if extended diagnostics are desired
0121	display-status	(n --)	Obsolete
0122	memory-test-suite	(addr len -- fail?)	Calls memory tester for given region
0124	mask	(-- a-addr)	Variable, holds "mask" used by memory-test-suite

Table 80 Time Utilities

Value	Function	Stack	Description
0125	get-msecs	(-- n)	Returns the current time, in milliseconds, approx.
0126	ms	(n --)	Delays for n milliseconds. Resolution is 1 millisecond
0213	alarm	(xt n --)	Periodically execute xt. If n=0, stop.

Table 81 Machine-specific Support

Value	Function	Stack	Description
0130	map-low	(phys.lo … size -- virt)	Maps a region of memory in 'sbus' address space
0131	sbus-intr>cpu	(sbus-intr# -- cpu-intr#)	Translates SBus interrupt# into CPU interrupt#

Note – Note – Table 82 through Table 89 apply only to `display` device-types.

Table 82 Terminal Emulator Interface

Value	Function	Stack	Description
011C	is-install	(xt --)	Identifies "install" routine to allocate a framebuffer
011D	is-remove	(xt --)	Identifies "remove" routine, to deallocate a framebuffer
011E	is-selftest	(xt --)	Identifies "selftest" routine for this framebuffer

Table 83 User-set Terminal Emulator State Values

Value	Function	Stack	Description
0150	#lines	(-- rows)	Number of lines of text being used for display. This word must be initialized (using `to`). `fb`*x*`-install` does this automatically, and also properly incorporates the configuration variable `screen-#rows`.
0151	#columns	(-- columns)	Number of columns (chars/line) used for display. This word must be initialized (using `to`). `fb`*x*`-install` does this automatically, and also properly incorporates the configuration variable `screen-#columns`.

Table 84 Terminal Emulator-set Terminal Emulator State Values

Value	Function	Stack	Description
0152	line#	(-- line#)	Current cursor position (line#). 0 is top line
0153	column#	(-- column#)	Current cursor position. 0 is left char.
0154	inverse?	(-- white-on-black?)	True if output is inverted (white-on-black)
0155	inverse-screen?	(-- black?)	True if screen has been inverted (black background)

Table 85 Display Device Low-level Interface `defer` Words

Value	Function	Stack	Description
0157	draw-character	(char --)	Paints the given character and advances the cursor
0158	reset-screen	(--)	Initializes the display device
0159	toggle-cursor	(--)	Draws or erase the cursor
015A	erase-screen	(--)	Clears all pixels on the display
015B	blink-screen	(--)	Flashes the display momentarily
015C	invert-screen	(--)	Changes all pixels to the opposite color
015D	insert-characters	(n --)	Inserts n blanks just before the cursor
015E	delete-characters	(n --)	Deletes n characters to the right of the cursor Remaining chars slide left
015F	insert-lines	(n --)	Inserts n blank lines just before the current line, lower lines are scrolled downward
0160	delete-lines	(n --)	Deletes n lines starting with the current line, lower lines are scrolled upward
0161	draw-logo	(line# addr width height --)	Draws the logo

Table 86 Frame Buffer Parameter Values*

Value	Function	Stack	Description
016C	char-height	(-- height)	Height (in pixels) of a character (usually 22)
016D	char-width	(-- width)	Width (in pixels) of a character (usually 12)
016F	fontbytes	(-- bytes)	Number of bytes/scan line for font entries (usually 2)

Table 86 Frame Buffer Parameter Values* *(Continued)*

Value	Function	Stack	Description
0162	frame-buffer-adr	(-- addr)	Address of frame buffer memory
0163	screen-height	(-- height)	Total height of the display (in pixels)
0164	screen-width	(-- width)	Total width of the display (in pixels)
0165	window-top	(-- border-height)	Distance (in pixels) between display top and text window
0166	window-left	(-- border-width)	Distance (in pixels) between display left edge and text window left edge
*These must all be initialized before using any `fb`*x*- routines.			

Table 87 Font Operators

Value	Function	Stack	Description
016A	default-font	(-- addr width height advance min- char #glyphs)	Returns default font values, plugs directly into set-font
016B	set-font	(addr width height advance min-char #glyphs --)	Sets the character font for text output
016E	>font	(char -- addr)	Returns font address for given ASCII character

Table 88 One-bit Framebuffer Utilities

Value	Function	Stack	Description
0170	fb1-draw-character	(char --)	Paints the character and advance the cursor
0171	fb1-reset-screen	(--)	Initializes the display device (noop)
0172	fb1-toggle-cursor	(--)	Draws or erases the cursor
0173	fb1-erase-screen	(--)	Clears all pixels on the display
0174	fb1-blink-screen	(--)	Inverts the screen, twice (slow)
0175	fb1-invert-screen	(--)	Changes all pixels to the opposite color
0176	fb1-insert-characters	(n --)	Inserts n blanks just before the cursor
0177	fb1-delete-characters	(n --)	Deletes n characters, starting at with cursor character, rightward. Remaining chars slide left
0178	fb1-insert-lines	(n --)	Inserts n blank lines just before the current line, lower lines are scrolled downward
0179	fb1-delete-lines	(n --)	Deletes n lines starting with the current line,lower lines are scrolled upward
017A	fb1-draw-logo	(line# addr width height --)	Draws the logo
017B	fb1-install	(width height #columns #lines --)	Installs the one-bit built-in routines
017C	fb1-slide-up	(n --)	Like fb1-delete-lines, but doesn't clear lines at bottom

Table 89 Eight-bit Framebuffer Utilities

Value	Function	Stack	Description
0180	fb8-draw-character	(char --)	Paints the character and advance the cursor
0181	fb8-reset-screen	(--)	Initializes the display device (noop)
0182	fb8-toggle-cursor	(--)	Draws or erases the cursor
0183	fb8-erase-screen	(--)	Clears all pixels on the display
0184	fb8-blink-screen	(--)	Inverts the screen, twice (slow)
0185	fb8-invert-screen	(--)	Changes all pixels to the opposite color
0186	fb8-insert-characters	(n --)	Inserts n blanks just before the cursor
0187	fb8-delete-characters	(n --)	Deletes n characters starting with cursor char, rightward. Remaining chars slide left
0188	fb8-insert-lines	(n --)	Inserts n blank lines just before the current line, lower lines are scrolled downward
0189	fb8-delete-lines	(n --)	Deletes n lines starting with the current line, lower lines are scrolled upward
018A	fb8-draw-logo	(line# addr width height --)	Draws the logo
018B	fb8-install	(width height #columns #lines --)	Installs the eight-bit built-in routines

Table 90 Package Support

Value	Function	Stack	Description
023C	peer	(phandle -- phandle.sibling)	Returns phandle of package that is the next child of the the parent of the package
023B	child	(phandle.parent -- phandle.child)	Returns phandle of the package that is the first child of the package parent-phandle
0204	find-package	(name-str name-len -- false | phandle true)	Finds a package named "name-str"
0205	open-package	(arg-str arg-len phandle -- ihandle | 0)	Opens an instance of the package "phandle," passes arguments "arg-str arg-len"
020F	$open-package	(arg-str arg-len name-str name-len -- ihandle | 0)	Finds a package "name-str name-len" then opens it with arguments "arg-str arg-len"
020A	my-parent	(-- ihandle)	Returns the ihandle of the parent of the current package instance
0203	my-self	(-- ihandle)	Returns the instance handle of currently-executing package instance
020B	ihandle>phandle	(ihandle -- phandle)	Converts an ihandle to a phandle
0206	close-package	(ihandle --)	Closes an instance of a package
0207	find-method	(method-str method-len phandle -- false | xt true)	Finds the method (command) named "method-str" within the package "phandle"
0208	call-package	(… xt ihandle -- ???)	Executes the method "xt" within the instance "ihandle"
020E	$call-method	(… method-str method-len ihandle -- ???)	Executes the method named "method-str" within the instance "ihandle"

Table 90 Package Support *(Continued)*

Value	Function	Stack	Description
0209	$call-parent	(… method-str method-len -- ???)	Executes the method "method-str" within the parent's package
0202	my-args	(-- arg-str arg-len)	Returns the argument str passed when this package was opened
020D	my-unit	(-- phys.lo … phys.hi)	Returns the physical unit number pair for this package
0102	my-address	(-- phys.lo …)	Returns the physical addr of this plug-in device. "phys" is a "magic" number, usable by other routines
0103	my-space	(-- phys.hi)	Returns address space of plug-in device. "space" is a "magic" number, usable by other routines

Table 91 Asynchronous Support

Value	Function	Stack	Description
0213	alarm	(xt n --)	Executes method (command) indicated by "xt" every "n" milliseconds
0219	user-abort	(… --) (R: … --)	Abort after alarm routine finishes execution

Table 92 Miscellaneous Operations

Value	Function	Stack	Description
0214	(is-user-word)	(E: … -- ???) (name-str name-len xt --)	Creates a new word called "name-str" which executes "xt"
01A4	mac-address	(-- mac-str mac-len)	Returns the MAC address

Table 93 Interpretation

Value	Function	Stack	Description
0215	suspend-fcode	(--)	Suspends execution of FCode, resumes later if an undefined command is required

Table 94 Error Handling

Value	Function	Stack	Description
0216	abort	(… --) (R:… --)	Aborts FCode execution, returns to the "ok" prompt
0217	catch	(… xt -- ??? error-code | ??? false)	Executes "xt," returns throw error code or 0 if throw not encountered
0218	throw	(… error-code -- ??? error-code | …)	Returns given error code to catch
FC	ferror	(--)	Displays "Unimplemented FCode" and stops FCode interpretation

FCodes by Byte Value

The following table lists, in hexadecimal order, currently-assigned FCode byte values. FCode values marked with an asterisk are available only on 64-bit implementations.

Table 95 FCodes by Byte Value

Value	Function	Stack
00	end0	(--)
10	b(lit)	(-- n) (F: /FCode-num32/ --)
11	b(')	(-- xt) (F: /FCode#/ --)
12	b(")	(-- str len) (F: /FCode-string/ --)
13	bbranch	(--) (F: /FCode-offset/ --)
14	b?branch	(don't-branch? --) (F: /FCode-offset/ --)
15	b(loop)	(--) (F: /FCode-offset/ --)
16	b(+loop)	(delta --) (F: /FCode-offset/ --)
17	b(do)	(limit start --) (F: /FCode-offset/ --)
18	b(?do)	(limit start --) (F: /FCode-offset/ --)
19	i	(-- index) (R: sys -- sys)
1A	j	(-- index) (R: sys -- sys)
1B	b(leave)	(F: --)
1C	b(of)	(sel of-val -- sel | <nothing>) (F: /FCode-offset/ --)
1D	execute	(… xt -- ???)
1E	+	(nu1 nu2 -- sum)
1F	-	(nu1 nu2 -- diff)
20	*	(nu1 nu2 -- prod)
21	/	(n1 n2 -- quot)
22	mod	(n1 n2 -- rem)
23	and	(x1 x2 -- x3)
24	or	(x1 x2 -- x3)
25	xor	(x1 x2 -- x3)
26	invert	(x1 -- x2)
27	lshift	(x1 u -- x2)
28	rshift	(x1 u -- x2)
29	>>a	(x1 u -- x2)
2A	/mod	(n1 n2 -- rem quot)
2B	u/mod	(u1 u2 -- urem uquot)
2C	negate	(n1 -- n2)
2D	abs	(n -- u)
2E	min	(n1 n2 -- n1|n2)
2F	max	(n1 n2 -- n1|n2)
30	>r	(x --) (R: -- x)
31	r>	(-- x) (R: x --)
32	r@	(-- x) (R: x -- x)
33	exit	(--) (R: sys --)

Table 95 FCodes by Byte Value *(Continued)*

Value	Function	Stack
34	0=	(nulflag -- equal-to-0?)
35	0<>	(n -- not-equal-to-0?)
36	0<	(n -- less-than-0?)
37	0<=	(n -- less-or-equal-to-0?)
38	0>	(n -- greater-than-0?)
39	0>=	(n -- greater-or-equal-to-0?)
3A	<	(n1 n2 -- less?)
3B	>	(n1 n2 -- greater?)
3C	=	(x1 x2 -- equal?)
3D	<>	(x1 x2 -- not-equal?)
3E	u>	(u1 u2 -- unsigned-greater?)
3F	u<=	(u1 u2 -- unsigned-less-or-equal?)
40	u<	(u1 u2 -- unsigned-less?)
41	u>=	(u1 u2 -- unsigned-greater-or-equal?)
42	>=	(n1 n2 -- greater-or-equal?)
43	<=	(n1 n2 -- less-or-equal?)
44	between	(n min max -- min<=n<=max?)
45	within	(n min max -- min<=n<max?)
46	drop	(x --)
47	dup	(x -- x x)
48	over	(x1 x2 -- x1 x2 x1)
49	swap	(x1 x2 -- x2 x1)
4A	rot	(x1 x2 x3 -- x2 x3 x1)
4B	-rot	(x1 x2 x3 -- x3 x1 x2)
4C	tuck	(x1 x2 -- x2 x1 x2)
4D	nip	(x1 x2 -- x2)
4E	pick	(xu … x1 x0 u -- xu … x1 x0 xu)
4F	roll	(xu … x1 x0 u -- xu-1 … x1 x0 xu)
50	?dup	(x -- 0 | x x)
51	depth	(-- u)
52	2drop	(x1 x2 --)
53	2dup	(x1 x2 -- x1 x2 x1 x2)
54	2over	(x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2)
55	2swap	(x1 x2 x3 x4 -- x3 x4 x1 x2)
56	2rot	(x1 x2 x3 x4 x5 x6 -- x3 x4 x5 x6 x1 x2)
57	2/	(x1 -- x2)
58	u2/	(x1 -- x2)
59	2*	(x1 -- x2)
5A	/c	(-- n)
5B	/w	(-- n)
5C	/l	(-- n)

Table 95 FCodes by Byte Value *(Continued)*

Value	**Function**	**Stack**
5D	/n	(-- n)
5E	ca+	(addr1 index -- addr2)
5F	wa+	(addr1 index -- addr2)
60	la+	(addr1 index -- addr2)
61	na+	(addr1 index -- addr2)
62	char+	(addr1 -- addr2)
63	wa1+	(addr1 -- addr2)
64	la1+	(addr1 -- addr2)
65	cell+	(addr1 -- addr2)
66	chars	(nu1 -- nu2)
67	/w*	(nu1 -- nu2)
68	/l*	(nu1 -- nu2)
69	cells	(nu1 -- nu2)
6A	on	(a-addr --)
6B	off	(a-addr --)
6C	+!	(nu a-addr --)
6D	@	(a-addr -- x)
6E	l@	(qaddr -- quad)
6F	w@	(waddr -- w)
70	<w@	(waddr -- n)
71	c@	(addr -- byte)
72	!	(x a-addr --)
73	l!	(quad qaddr --)
74	w!	(w waddr --)
75	c!	(byte addr --)
76	2@	(a-addr -- x1 x2)
77	2!	(x1 x2 a-addr --)
78	move	(src-addr dest-addr len --)
79	fill	(addr len byte --)
7A	comp	(addr1 addr2 len -- n)
7B	noop	(--)
7C	lwsplit	(quad -- w1.lo w2.hi)
7D	wljoin	(w.lo w.hi -- quad)
7E	lbsplit	(quad -- b.lo b2 b3 b4.hi)
7F	bljoin	(bl.lo b2 b3 b4.hi -- quad)
80	wbflip	(w1 -- w2)
81	upc	(char1 -- char2)
82	lcc	(char1 -- char2)
83	pack	(str len addr -- pstr)
84	count	(pstr -- str len)
85	body>	(a-addr -- xt)

Table 95 FCodes by Byte Value *(Continued)*

Value	**Function**	**Stack**
86	>body	(xt -- a-addr)
87	fcode-revision	(-- n)
88	span	(-- a-addr)
89	unloop	(--) (R: sys --)
8A	expect	(addr len --)
8B	alloc-mem	(len -- a-addr)
8C	free-mem	(a-addr len --)
8D	key?	(-- pressed?)
8E	key	(-- char)
8F	emit	(char --)
90	type	(text-str text-len --)
91	(cr	(--)
92	cr	(--)
93	#out	(-- a-addr)
94	#line	(-- a-addr)
95	hold	(char --)
96	<#	(--)
97	u#>	(u -- str len)
98	sign	(n --)
99	u#	(u1 -- u2)
9A	u#s	(u1 -- u2)
9B	u.	(u --)
9C	u.r	(u size --)
9D	.	(nu --)
9E	.r	(n size --)
9F	.s	(… -- …)
A0	base	(-- a-addr)
A2	$number	(addr len -- true | n false)
A3	digit	(char base -- digit true | char false)
A4	-1	(-- -1)
A5	0	(-- 0)
A6	1	(-- 1)
A7	2	(-- 2)
A8	3	(-- 3)
A9	bl	(-- 0x20)
AA	bs	(-- 0x08)
AB	bell	(-- 0x07)
AC	bounds	(n cnt -- n+cnt n)
AD	here	(-- addr)
AE	aligned	(n1 -- n1|a-addr)
AF	wbsplit	(w -- b1.lo b2.hi)

Table 95 FCodes by Byte Value *(Continued)*

Value	**Function**	**Stack**
B0	bwjoin	(b.lo b.hi -- w)
B1	b(<mark)	(F: --)
B2	b(>resolve)	(--) (F: --)
B5	new-token	(--) (F: /FCode#/ --)
B6	named-token	(--) (F: /FCode-string FCode#/ --)
B7	b(:)	(E: ... -- ???) (F: -- colon-sys)
B8	b(value)	(E: -- x) (F: x --)
B9	b(variable)	(E: -- a-addr) (F: --)
BA	b(constant)	(E: -- n) (F: n --)
BB	b(create)	(E: -- a-addr) (F: --)
BC	b(defer)	(E: ... -- ???) (F: --)
BD	b(buffer:)	(E: -- a-addr) (F: size --)
BE	b(field)	(E: addr -- addr+offset) (F: offset size -- offset+size)
C0	instance	(--)
C2	b(;)	(--) (F: colon-sys --)
C3	b(to)	(x --)
C4	b(case)	(sel -- sel) (F: --)
C5	b(endcase)	(sel --) (F: --)
C6	b(endof)	(--) (F: /FCode-offset/ --)
C7	#	(ud1 -- ud2)
C8	#s	(ud -- 0 0)
C9	#>	(ud -- str len)
CA	external-token	(--) (F: /FCode-string FCode#/ --)
CB	$find	(name-str name-len -- xt true | name-str name-len false)
CC	offset16	(--)
CD	eval	(... str len -- ???)
D0	c,	(byte --)
D1	w,	(w --)
D2	l,	(quad --)
D3	,	(x --)
D4	um*	(u1 u2 -- ud.prod)
D5	um/mod	(ud u -- urem uquot)
D8	d+	(d1 d2 --d.sum)
D9	d-	(d1 d2 -- d.diff)
DA	get-token	(fcode# -- xt immediate?)
DB	set-token	(xt immediate? fcode# --)
DC	state	(-- a-addr)
DD	compile,	(xt --)
DE	behavior	(defer-xt -- contents-xt)
F0	start0	(--)
F1	start1	(--)

Table 95 FCodes by Byte Value *(Continued)*

Value	Function	Stack
F2	start2	(--)
F3	start4	(--)
FC	ferror	(--)
FD	version1	(--)
FF	end1	(--)
0102	my-address	(-- phys.lo …)
0103	my-space	(-- phys.hi)
0105	free-virtual	(virt size --)
0110	property	(prop-addr prop-len name-str name-len --)
0111	encode-int	(n -- prop-addr prop-len)
0112	encode+	(prop-addr1 prop-len1 prop-addr2 prop-len2 -- prop-addr3 prop-len3)
0113	encode-phys	(phys.lo … phys.hi -- prop-addr prop-len)
0114	encode-string	(str len -- prop-addr prop-len)
0115	encode-bytes	(data-addr data-len -- prop-addr prop-len)
0116	reg	(phys.lo … phys.hi size --)
0119	model	(str len --)
011A	device-type	(str len --)
011B	parse-2int	(str len -- val.lo val.hi)
011C	is-install	(xt --)
011D	is-remove	(xt --)
011E	is-selftest	(xt --)
011F	new-device	(--)
0120	diagnostic-mode?	(-- diag?)
0121	display-status	(n --)
0122	memory-test-suite	(addr len -- fail?)
0124	mask	(-- a-addr)
0125	get-msecs	(-- n)
0126	ms	(n --)
0127	finish-device	(--)
0128	decode-phys	(prop-addr1 prop-len1 -- prop-addr2 prop-len2 phys.lo … phys.hi)
0130	map-low	(phys.lo … size -- virt)
0131	sbus-intr>cpu	(sbus-intr# -- cpu-intr#)
0150	#lines	(-- rows)
0151	#columns	(-- columns)
0152	line#	(-- line#)
0153	column#	(-- column#)
0154	inverse?	(-- white-on-black?)
0155	inverse-screen?	(-- black?)
0157	draw-character	(char --)
0158	reset-screen	(--)
0159	toggle-cursor	(--)

Table 95 FCodes by Byte Value *(Continued)*

Value	Function	Stack
015A	erase-screen	(--)
015B	blink-screen	(--)
015C	invert-screen	(--)
015D	insert-characters	(n --)
015E	delete-characters	(n --)
015F	insert-lines	(n --)
0160	delete-lines	(n --)
0161	draw-logo	(line# addr width height --)
0162	frame-buffer-adr	(-- addr)
0163	screen-height	(-- height)
0164	screen-width	(-- width)
0165	window-top	(-- border-height)
0166	window-left	(-- border-width)
016A	default-font	(-- addr width height advance min-char #glyphs)
016B	set-font	(addr width height advance min-char #glyphs --)
016C	char-height	(-- height)
016D	char-width	(-- width)
016E	>font	(char -- addr)
016F	fontbytes	(-- bytes)
0170	fb1-draw-character	(char --)
0171	fb1-reset-screen	(--)
0172	fb1-toggle-cursor	(--)
0173	fb1-erase-screen	(--)
0174	fb1-blink-screen	(--)
0175	fb1-invert-screen	(--)
0176	fb1-insert-characters	(n --)
0177	fb1-delete-characters	(n --)
0178	fb1-insert-lines	(n --)
0179	fb1-delete-lines	(n --)
017A	fb1-draw-logo	(line# addr width height --)
017B	fb1-install	(width height #columns #lines --)
017C	fb1-slide-up	(n --)
0180	fb8-draw-character	(char --)
0181	fb8-reset-screen	(--)
0182	fb8-toggle-cursor	(--)
0183	fb8-erase-screen	(--)
0184	fb8-blink-screen	(--)
0185	fb8-invert-screen	(--)
0186	fb8-insert-characters	(n --)
0187	fb8-delete-characters	(n --)
0188	fb8-insert-lines	(n --)

Table 95 FCodes by Byte Value *(Continued)*

Value	**Function**	**Stack**
0189	fb8-delete-lines	(n --)
018A	fb8-draw-logo	(line# addr width height --)
018B	fb8-install	(width height #columns #lines --)
01A4	mac-address	(-- mac-str mac-len)
0201	device-name	(str len --)
0202	my-args	(-- arg-str arg-len)
0203	my-self	(-- ihandle)
0204	find-package	(name-str name-len -- false | phandle true)
0205	open-package	(arg-str arg-len phandle -- ihandle | 0)
0206	close-package	(ihandle --)
0207	find-method	(method-str method-len phandle -- false | xt true)
0208	call-package	(… xt ihandle -- ???)
0209	$call-parent	(… method-str method-len -- ???)
020A	my-parent	(-- ihandle)
020B	ihandle>phandle	(ihandle -- phandle)
020D	my-unit	(-- phys.lo … phys.hi)
020E	$call-method	(… method-str method-len ihandle -- ???)
020F	$open-package	(arg-str arg-len name-str name-len -- ihandle | 0)
0213	alarm	(xt n --)
0214	(is-user-word)	(E: … -- ???) (name-str name-len xt --)
0215	suspend-fcode	(--)
0216	abort	(… --) (R:… --)
0217	catch	(… xt -- ??? error-code | ??? false)
0218	throw	(… error-code -- ??? error-code | …)
0219	user-abort	(… --) (R: … --)
021A	get-my-property	(nam-str nam-len -- true | prop-addr prop-len false)
021B	decode-int	(prop-addr1 prop-len1 -- prop-addr2 prop-len2 n)
021C	decode-string	(prop-addr1 prop-len1 -- prop-addr2 prop-len2 str len)
021D	get-inherited-property	(nam-str nam-len -- true | prop-addr prop-len false)
021E	delete-property	(nam-str nam-len --)
021F	get-package-property	(name-str name-len phandle -- true | prop-addr prop-len false)
0220	cpeek	(addr -- false | byte true)
0221	wpeek	(waddr -- false | w true)
0222	lpeek	(qaddr -- false | quad true)
0223	cpoke	(byte addr -- okay?)
0224	wpoke	(w waddr -- okay?)
0225	lpoke	(quad qaddr -- okay?)
0226	lwflip	(quad1 -- quad2)
0227	lbflip	(quad1 -- quad2)
0228	lbflips	(qaddr len --)
022E*	rx@	(oaddr -- o)

Table 95 FCodes by Byte Value *(Continued)*

Value	**Function**	**Stack**
022F*	rx!	(o oaddr --)
0230	rb@	(addr -- byte)
0231	rb!	(byte addr --)
0232	rw@	(waddr -- w)
0233	rw!	(w waddr --)
0234	rl@	(qaddr -- quad)
0235	rl!	(quad qaddr --)
0236	wbflips	(waddr len --)
0237	lwflips	(qaddr len --)
023B	child	(phandle.parent -- phandle.child)
023C	peer	(phandle -- phandle.sibling)
023D	next-property	(previous-str previous-len phandle -- false | name-str name-len true)
023E	byte-load	(addr xt --)
023F	set-args	(arg-str arg-len unit-str unit-len --)
0240	left-parse-string	(str len char -- R-str R-len L-str L-len)
0241*	bxjoin	(b.lo b.2 b.3 b.4 b.5 b.6 b.7 b.hi -- o)
0242*	<l@	(qaddr -- n)
0243*	lxjoin	(quad.lo quad.hi -- o)
0244*	wxjoin	(w.lo w.2 w.3 w.hi -- o)
0245*	x,	(o --)
0246*	x@	(oaddr -- o)
0247*	x!	(o oaddr --)
0248*	/x	(-- n)
0249*	/x*	(nu1 -- nu2)
024A*	xa+	(addr1 index -- addr2)
024B*	xa1+	(addr1 -- addr2)
024C*	xbflip	(oct1 -- oct2)
024D*	xbflips	(oaddr len --)
024E*	xbsplit	(o -- b.lo b.2 b.3 b.4 b.5 b.6 b.7 b.hi)
024F*	xlflip	(oct1 -- oct2)
0250*	xlflips	(oaddr len --)
0251*	xlsplit	(o -- quad.lo quad.hi)
0252*	xwflip	(oct1 -- oct2)
0253*	xwflips	(oaddr len --)
0254*	xwsplit	(o -- w.lo w.2 w.3 w.hi)
-	(	([text<)> --)
-	]tokenizer	(--)
-	\	(--)
-	alias	(E: … -- ???) (“new-name< >old-name< >” --)
-	decimal	(--)

Table 95 FCodes by Byte Value *(Continued)*

Value	**Function**	**Stack**
-	external	(--)
-	fload	([filename<cr>] --)
-	headerless	(--)
-	headers	(--)
-	hex	(--)
-	octal	(--)
-	tokenizer[	(--)
TG	"	([text<">< >] -- text-str text-len)
TG	'	("old-name< >" -- xt)
TG	(.)	(n -- str len)
TG	."	([text<)>] --)
TG	.(	([text<)>] --)
TG	: (colon)	("new-name< >" -- colon-sys) (E: … -- ???)
TG	; (semicolon)	(--)
TG	<<	(x1 u -- x2)
TG	>>	(x1 u -- x2)
TG	?	(addr --)
TG	[']	([old-name< >] -- xt)
TG	1+	(nu1 -- nu2)
TG	1-	(nu1 -- nu2)
TG	2+	(nu1 -- nu2)
TG	2-	(nu1 -- nu2)
TG	accept	(addr len1 -- len2)
TG	again	(C: dest-sys --)
TG	allot	(len --)
TG	ascii	([text< >] -- char)
TG	begin	(C: -- dest-sys) (--)
TG	blank	(addr len --)
TG	buffer:	(E: -- a-addr) (len "new-name< >" --)
TG	/c*	(nu1 -- nu2)
TG	ca1+	(addr1 -- addr2)
TG	carret	(-- 0x0D)
TG	case	(C: -- case-sys) (sel -- sel)
TG	constant	(E: -- x) (x "new-name< >" --)
TG	control	([text< >] -- char)
TG	create	(E: -- a-addr) ("new-name< >" --)
TG	d#	([number< >] -- n)
TG	.d	(n --)
TG	decimal	(--)
TG	decode-bytes	(prop-addr1 prop-len1 data-len -- prop-addr2 prop-len2 data-addr data-len)
TG	defer	(E: … -- ???) ("new-name< >" --)

Table 95 FCodes by Byte Value *(Continued)*

Value	Function	Stack
TG	do	(C: -- dodest-sys) (limit start --) (R: -- sys)
TG	?do	(C: -- dodest-sys) (limit start --) (R: -- sys)
TG	3drop	(x1 x2 x3 --)
TG	3dup	(x1 x2 x3 -- x1 x2 x3 x1 x2 x3)
TG	else	(C: orig-sys1 -- orig-sys2) (--)
TG	endcase	(C: case-sys --) (sel --)
TG	endof	(C: case-sys1 of-sys -- case-sys2) (--)
TG	erase	(addr len --)
TG	eval	(… str len -- ???)
TG	false	(-- false)
TG	fcode-version2	(--)
TG	field	(E: addr -- addr+offset) (offset size "new-name< >" -- offset+size)
TG	h#	([number< >] -- n)
TG	.h	(n --)
TG	hex	(--)
TG	if	(C: -- orig-sys) (do-next? --)
TG	leave	(--) (R: sys --)
TG	?leave	(exit? --) (R: sys --)
TG	linefeed	(-- 0x0A)
TG	loop	(C: dodest-sys --) (--) (R: sys1 -- <nothing> | sys2)
TG	+loop	(C: dodest-sys --) (delta --) (R: sys1 -- <nothing> | sys2)
TG	/n*	(nu1 -- nu2)
TG	na1+	(addr1 -- addr2)
TG	not	(x1 -- x2)
TG	o#	([number< >] -- n)
TG	octal	(--)
TG	of	(C: case-sys1 -- case-sys2 of-sys) (sel of-val -- sel | <nothing>)
TG	repeat	(C: orig-sys dest-sys --) (--)
TG	s"	([text<“>] -- test-str text-len)
TG	s.	(n --)
TG	space	(--)
TG	spaces	(cnt --)
TG	struct	(-- 0)
TG	then	(C: orig-sys --) (--)
TG	to	(param [old-name< >] --)
TG	true	(-- true)
TG	(u.)	(u -- str len)
TG	until	(C: dest-sys --) (done? --)
TG	value	(E: -- x) (x "new-name< >"--)
TG	variable	(E: -- a-addr) ("new-name< >"--)
TG	while	(C: dest-sys -- orig-sys dest-sys) (continue? --)

FCodes by Name

The following table lists, in alphabetic order, currently-assigned FCodes. FCode values marked with an asterisk are available only on 64-bit implementations.

Table 96 FCodes by Name

Value	Function	Stack
72	!	(x a-addr --)
TG	"	([text<">< >] -- text-str text-len)
C7	#	(ud1 -- ud2)
C9	#>	(ud -- str len)
TG	'	("old-name< >" -- xt)
-	(	([text<)> --)
TG	(.)	(n -- str len)
20	*	(nu1 nu2 -- prod)
1E	+	(nu1 nu2 -- sum)
6C	+!	(nu a-addr --)
D3	,	(x --)
1F	-	(nu1 nu2 -- diff)
9D	.	(nu --)
TG	."	([text<)>] --)
TG	.(	([text<)>] --)
21	/	(n1 n2 -- quot)
TG	: (colon)	("new-name< >" -- colon-sys) (E: ... -- ???)
TG	; (semicolon)	(--)
3A	<	(n1 n2 -- less?)
96	<#	(--)
TG	<<	(x1 u -- x2)
43	<=	(n1 n2 -- less-or-equal?)
3D	<>	30
3C	=	(n1 n2 -- greater?)
0B	>	(n1 n2 -- greater?)
42	>=	(n1 n2 -- greater-or-equal?)
TG	>>	(x1 u -- x2)
TG	?	(addr --)
6D	@	(a-addr -- x)
TG	[']	([old-name< >] -- xt)
-	\	(--)
-	]tokenizer	(--)
A5	0	(-- 0)
36	0<	(n -- less-than-0?)
37	0<=	(n -- less-or-equal-to-0?)
35	0<>	(n -- not-equal-to-0?)
34	0=	(nulflag -- equal-to-0?)

Table 96 FCodes by Name *(Continued)*

Value	Function	Stack
38	0>	(n -- greater-than-0?)
39	0>=	(n -- greater-or-equal-to-0?)
A6	1	(-- 1)
TG	1+	(nu1 -- nu2)
TG	1-	(nu1 -- nu2)
A4	-1	(-- -1)
A7	2	(-- 2)
77	2!	(x1 x2 a-addr --)
59	2*	(x1 -- x2)
TG	2+	(nu1 -- nu2)
TG	2-	(nu1 -- nu2)
57	2/	(x1 -- x2)
76	2@	(a-addr -- x1 x2)
A8	3	(-- 3)
29	>>a	(x1 u -- x2)
0216	abort	(… --) (R:… --)
2D	abs	(n -- u)
TG	accept	(addr len1 -- len2)
TG	again	(C: dest-sys --)
0213	alarm	(xt n --)
-	alias	(E: … -- ???) (“new-name< >old-name< >” --)
AE	aligned	(n1 -- n1|a-addr)
8B	alloc-mem	(len -- a-addr)
TG	allot	(len --)
23	and	(x1 x2 -- x3)
TG	ascii	([text< >] -- char)
12	b(")	(-- str len) (F: /FCode-string/ --)
11	b(')	(-- xt) (F: /FCode#/ --)
B7	b(:)	(E: … -- ???) (F: -- colon-sys)
C2	b(;)	(--) (F: colon-sys --)
A0	base	(-- a-addr)
13	g	(--) (F: /FCode-offset/ --)
14	b?branch	(don’t-branch? --) (F: /FCode-offset/ --)
BD	b(buffer:)	(E: -- a-addr) (F: size --)
C4	b(case)	(sel -- sel) (F: --)
BA	b(constant)	(E: -- n) (F: n --)
BB	b(create)	(E: -- a-addr) (F: --)
BC	b(defer)	(E: … -- ???) (F: --)
17	b(do)	(limit start --) (F: /FCode-offset/ --)
18	b(?do)	(limit start --) (F: /FCode-offset/ --)
TG	begin	(C: -- dest-sys) (--)

Table 96 FCodes by Name *(Continued)*

Value	Function	Stack
DE	behavior	(defer-xt -- contents-xt)
AB	bell	(-- 0x07)
C5	b(endcase)	(sel --) (F: --)
C6	b(endof)	(--) (F: /FCode-offset/ --)
44	between	(n min max -- min<=n<=max?)
BE	b(field)	(E: addr -- addr+offset) (F: offset size -- offset+size)
A9	bl	(-- 0x20)
TG	blank	(addr len --)
1B	b(leave)	(F: --)
015B	blink-screen	(--)
10	b(lit)	(-- n) (F: /FCode-num32/ --)
7F	bljoin	(bl.lo b2 b3 b4.hi -- quad)
15	b(loop)	(--) (F: /FCode-offset/ --)
16	b(+loop)	(delta --) (F: /FCode-offset/ --)
B1	b(<mark)	(F: --)
85	body>	(a-addr -- xt)
86	>body	(xt -- a-addr)
1C	b(of)	(sel of-val -- sel | <nothing>) (F: /FCode-offset/ --)
AC	bounds	(n cnt -- n+cnt n)
B2	b(>resolve)	(--) (F: --)
AA	bs	(-- 0x08)
C3	b(to)	(x --)
TG	buffer:	(E: -- a-addr) (len "new-name< >" --)
B8	b(value)	(E: -- x) (F: x --)
B9	b(variable)	(E: -- a-addr) (F: --)
B0	bwjoin	(b.lo b.hi -- w)
241*	bxjoin	(b.lo b.2 b.3 b.4 b.5 b.6 b.7 b.hi -- o)
023E	byte-load	(addr xt --)
75	c!	(byte addr --)
D0	c,	(byte --)
5A	/c	(-- n)
-	/c*	(nu1 -- nu2)
71	c@	(addr -- byte)
5E	ca+	(addr1 index -- addr2)
TG	ca1+	(addr1 -- addr2)
62	char+	(addr1 -- addr2)
020E	$call-method	(… method-str method-len ihandle -- ???)
0208	call-package	(… xt ihandle -- ???)
0209	$call-parent	(… method-str method-len -- ???)
TG	carret	(-- 0x0D)
TG	case	(C: -- case-sys) (sel -- sel)

Table 96 FCodes by Name *(Continued)*

Value	**Function**	**Stack**
0217	catch	(… xt -- ??? error-code | ??? false)
65	cell+	(addr1 -- addr2)
69	cells	(nu1 -- nu2)
62	char+	(addr1 -- addr2)
016C	char-height	(-- height)
66	chars	(nu1 -- nu2)
016D	char-width	(-- width)
0236	child	(phandle.parent -- phandle.child)
0206	close-package	(ihandle --)
0153	column#	(-- column#)
0151	#columns	(-- columns)
7A	comp	(addr1 addr2 len -- n)
DD	compile,	(xt --)
TG	constant	(E: -- x) (x "new-name< >" --)
TG	control	([text< >] -- char)
84	count	(pstr -- str len)
0220	cpeek	(addr -- false | byte true)
0223	cpoke	(byte addr -- okay?)
92	cr	(--)
91	(cr	(--)
TG	create	(E: -- a-addr) ("new-name< >" --)
TG	d#	([number< >] -- n)
D8	d+	(d1 d2 --d.sum)
D9	d-	(d1 d2 -- d.diff)
TG	.d	(n --)
-	decimal	(--)
TG	decimal	(--)
021B	decode-int	(prop-addr1 prop-len1 -- prop-addr2 prop-len2 n)
0128	decode-phys	(prop-addr1 prop-len1 -- prop-addr2 prop-len2 phys.lo … phys.hi)
021C	decode-string	(prop-addr1 prop-len1 -- prop-addr2 prop-len2 str len)
016A	default-font	(-- addr width height advance min-char #glyphs)
TG	defer	(E: … -- ???) ("new-name< >" --)
015E	delete-characters	(n --)
0160	delete-lines	(n --)
021E	delete-property	(nam-str nam-len --)
51	depth	(-- u)
0201	device-name	(str len --)
011A	device-type	(str len --)
0120	diagnostic-mode?	(-- diag?)
A3	digit	(char base -- digit true | char false)
0121	display-status	(n --)

Table 96 FCodes by Name *(Continued)*

Value	Function	Stack
TG	do	(C: -- dodest-sys) (limit start --) (R: -- sys)
TG	?do	(C: -- dodest-sys) (limit start --) (R: -- sys)
0157	draw-character	(char --)
0161	draw-logo	(line# addr width height --)
46	drop	(x --)
52	2drop	(x1 x2 --)
TG	3drop	(x1 x2 x3 --)
47	dup	(x -- x x)
53	2dup	(x1 x2 -- x1 x2 x1 x2)
TG	3dup	(x1 x2 x3 -- x1 x2 x3 x1 x2 x3)
50	?dup	(x -- 0 | x x)
TG	else	(C: orig-sys1 -- orig-sys2) (--)
8F	emit	(char --)
0112	encode+	(prop-addr1 prop-len1 prop-addr2 prop-len2 -- prop-addr3 prop-len3)
0115	encode-bytes	(data-addr data-len -- prop-addr prop-len)
0111	encode-int	(n -- prop-addr prop-len)
0113	encode-phys	(phys.lo … phys.hi -- prop-addr prop-len)
0114	encode-string	(str len -- prop-addr prop-len)
00	end0	(--)
FF	end1	(--)
TG	endcase	(C: case-sys --) (sel --)
TG	endof	(C: case-sys1 of-sys -- case-sys2) (--)
TG	erase	(addr len --)
015A	erase-screen	(--)
TG	eval	(… str len -- ???)
CD	evaluate	(… str len -- ???)
1D	execute	(… xt -- ???)
33	exit	(--) (R: sys --)
8A	expect	(addr len --)
-	external	(--)
CA	external-token	(--) (F: /FCode-string FCode#/ --)
TG	false	(-- false)
0174	fb1-blink-screen	(--)
0177	fb1-delete-characters	(n --)
0179	fb1-delete-lines	(n --)
0170	fb1-draw-character	(char --)
017A	fb1-draw-logo	(line# addr width height --)
0173	fb1-erase-screen	(--)
0176	fb1-insert-characters	(n --)
0178	fb1-insert-lines	(n --)
017B	fb1-install	(width height #columns #lines --)

Table 96 FCodes by Name *(Continued)*

Value	Function	Stack
0175	fb1-invert-screen	(--)
0171	fb1-reset-screen	(--)
017C	fb1-slide-up	(n --)
0172	fb1-toggle-cursor	(--)
0184	fb8-blink-screen	(--)
0187	fb8-delete-characters	(n --)
0189	fb8-delete-lines	(n --)
0180	fb8-draw-character	(char --)
018A	fb8-draw-logo	(line# addr width height --)
0183	fb8-erase-screen	(--)
0186	fb8-insert-characters	(n --)
0188	fb8-insert-lines	(n --)
018B	fb8-install	(width height #columns #lines --)
0185	fb8-invert-screen	(--)
0181	fb8-reset-screen	(--)
0182	fb8-toggle-cursor	(--)
87	fcode-revision	(-- n)
TG	fcode-version2	(--)
FC	ferror	(--)
TG	field	(E: addr -- addr+offset) (offset size "new-name< >" -- offset+size)
79	fill	(addr len byte --)
CB	$find	(name-str name-len -- xt true | name-str name-len false)
0207	find-method	(method-str method-len phandle -- false | xt true)
0204	find-package	(name-str name-len -- false | phandle true)
0127	finish-device	(--)
016E	>font	(char -- addr)
-	fload	([filename<cr>] --)
016F	fontbytes	(-- bytes)
0162	frame-buffer-adr	(-- addr)
8C	free-mem	(a-addr len --)
0105	free-virtual	(virt size --)
021d	get-inherited-property	(nam-str nam-len -- true | prop-addr prop-len false)
0125	get-msecs	(-- n)
021A	get-my-property	(nam-str nam-len -- true | prop-addr prop-len false)
021F	get-package-property	(name-str name-len phandle -- true | prop-addr prop-len false)
DA	get-token	(fcode# -- xt immediate?)
TG	h#	([number< >] -- n)
TG	.h	(n --)
-	headerless	(--)
-	headers	(--)
AD	here	(-- addr)

Table 96 FCodes by Name *(Continued)*

Value	Function	Stack
-	hex	(--)
TG	hex	(--)
95	hold	(char --)
19	i	(-- index) (R: sys -- sys)
TG	if	(C: -- orig-sys) (do-next? --)
020B	ihandle>phandle	(ihandle -- phandle)
015D	insert-characters	(n --)
015F	insert-lines	(n --)
C0	instance	(--)
0154	inverse?	(-- white-on-black?)
0155	inverse-screen?	(-- black?)
26	invert	(x1 -- x2)
015C	invert-screen	(--)
011C	is-install	(xt --)
011D	is-remove	(xt --)
011E	is-selftest	(xt --)
0214	(is-user-word)	(E: ... -- ???) (name-str name-len xt --)
1A	j	(-- index) (R: sys -- sys)
8E	key	(-- char)
8D	key?	(-- pressed?)
73	l!	(quad qaddr --)
D2	l,	(quad --)
6E	l@	(qaddr -- quad)
5C	/l	(-- n)
68	/l*	(nu1 -- nu2)
242*	<l@	(qaddr -- n)
60	la+	(addr1 index -- addr2)
64	la1+	(addr1 -- addr2)
0227	lbflip	(quad1 -- quad2)
0228	lbflips	(qaddr len --)
7E	lbsplit	(quad -- b.lo b2 b3 b4.hi)
82	lcc	(char1 -- char2)
TG	leave	(--) (R: sys --)
TG	?leave	(exit? --) (R: sys --)
0240	left-parse-string	(str len char -- R-str R-len L-str L-len)
0152	line#	(-- line#)
94	#line	(-- a-addr)
TG	linefeed	(-- 0x0A)
0150	#lines	(-- rows)
TG	loop	(C: dodest-sys --) (--) (R: sys1 -- <nothing> | sys2)
TG	+loop	(C: dodest-sys --) (delta --) (R: sys1 -- <nothing> | sys2)

Table 96 FCodes by Name *(Continued)*

Value	**Function**	**Stack**
0222	lpeek	(qaddr -- false | quad true)
0225	lpoke	(quad qaddr -- okay?)
27	lshift	(x1 u -- x2)
0226	lwflip	(quad1 -- quad2)
0237	lwflips	(qaddr len --)
7C	lwsplit	(quad -- w1.lo w2.hi)
243*	lxjoin	(quad.lo quad.hi -- o)
01A4	mac-address	(-- mac-str mac-len)
0130	map-low	(phys.lo … size -- virt)
0124	mask	(-- a-addr)
2F	max	(n1 n2 -- n1|n2)
0122	memory-test-suite	(addr len -- fail?)
2E	min	(n1 n2 -- n1|n2)
22	mod	(n1 n2 -- rem)
2A	/mod	(n1 n2 -- rem quot)
0119	model	(str len --)
78	move	(src-addr dest-addr len --)
0126	ms	(n --)
0102	my-address	(-- phys.lo …)
0202	my-args	(-- arg-str arg-len)
020A	my-parent	(-- ihandle)
0203	my-self	(-- ihandle)
0103	my-space	(-- phys.hi)
020D	my-unit	(-- phys.lo … phys.hi)
5D	/n	(-- n)
TG	/n*	(nu1 -- nu2)
61	na+	(addr1 index -- addr2)
TG	na1+	(addr1 -- addr2)
B6	named-token	(--) (F: /FCode-string FCode#/ --)
2C	negate	(n1 -- n2)
011F	new-device	(--)
B5	new-token	(--) (F: /FCode#/ --)
023D	next-property	(previous-str previous-len phandle -- false | name-str name-len true)
4D	nip	(x1 x2 -- x2)
7B	noop	(--)
TG	not	(x1 -- x2)
A2	$number	(addr len -- true | n false)
TG	o#	([number< >] -- n)
-	octal	(--)
TG	octal	(--)
TG	of	(C: case-sys1 -- case-sys2 of-sys) (sel of-val -- sel | <nothing>)

Table 96 FCodes by Name *(Continued)*

Value	**Function**	**Stack**
6B	off	(a-addr --)
CC	offset16	(--)
6A	on	(a-addr --)
0205	open-package	(arg-str arg-len phandle -- ihandle | 0)
020F	$open-package	(arg-str arg-len name-str name-len -- ihandle | 0)
24	or	(x1 x2 -- x3)
93	#out	(-- a-addr)
48	over	(x1 x2 -- x1 x2 x1)
54	2over	(x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2)
83	pack	(str len addr -- pstr)
011B	parse-2int	(str len -- val.lo val.hi)
023C	peer	(phandle -- phandle.sibling)
4E	pick	(xu … x1 x0 u -- xu … x1 x0 xu)
0110	property	(prop-addr prop-len name-str name-len --)
31	r>	(-- x) (R: x --)
32	r@	(-- x) (R: x -- x)
9E	.r	(n size --)
30	>r	(x --) (R: -- x)
0231	rb!	(byte addr --)
0230	rb@	(addr -- byte)
0116	reg	(phys.lo … phys.hi size --)
TG	repeat	(C: orig-sys dest-sys --) (--)
0158	reset-screen	(--)
0235	rl!	(quad qaddr --)
0234	rl@	(qaddr -- quad)
4F	roll	(xu … x1 x0 u -- xu-1 … x1 x0 xu)
4A	rot	(x1 x2 x3 -- x2 x3 x1)
4B	-rot	(x1 x2 x3 -- x3 x1 x2)
56	2rot	(x1 x2 x3 x4 x5 x6 -- x3 x4 x5 x6 x1 x2)
28	rshift	(x1 u -- x2)
0233	rw!	(w waddr --)
0232	rw@	(waddr -- w)
022E*	rx@	(oaddr -- o)
022F*	rx!	(o oaddr --)
TG	s"	([text<“>] -- test-str text-len)
TG	s.	(n --)
C8	#s	(ud -- 0 0)
9F	.s	(… -- …)
0131	sbus-intr>cpu	(sbus-intr# -- cpu-intr#)
0163	screen-height	(-- height)

Table 96 FCodes by Name *(Continued)*

Value	Function	Stack
0164	screen-width	(-- width)
023F	set-args	(arg-str arg-len unit-str unit-len --)
016B	set-font	(addr width height advance min-char #glyphs --)
DB	set-token	(xt immediate? fcode# --)
98	sign	(n --)
TG	space	(--)
TG	spaces	(cnt --)
88	span	(-- a-addr)
F0	start0	(--)
F1	start1	(--)
F2	start2	(--)
F3	start4	(--)
DC	state	(-- a-addr)
TG	struct	(-- 0)
0215	suspend-fcode	(--)
49	swap	(x1 x2 -- x2 x1)
55	2swap	(x1 x2 x3 x4 -- x3 x4 x1 x2)
TG	then	(C: orig-sys --) (--)
0218	throw	(… error-code -- ??? error-code | …)
TG	to	(param [old-name< >] --)
0159	toggle-cursor	(--)
-	tokenizer[	(--)
TG	true	(-- true)
4C	tuck	(x1 x2 -- x2 x1 x2)
90	type	(text-str text-len --)
99	u#	(u1 -- u2)
97	u#>	(u -- str len)
9A	u#s	(u1 -- u2)
9B	u.	(u --)
40	u<	(u1 u2 -- unsigned-less?)
3F	u<=	(u1 u2 -- unsigned-less-or-equal?)
3E	u>	(u1 u2 -- unsigned-greater?)
41	u>=	(u1 u2 -- unsigned-greater-or-equal?)
TG	(u.)	(n -- addr len)
58	u2/	(x1 -- x2)
D4	um*	(u1 u2 -- ud.prod)
D5	um/mod	(ud u -- urem uquot)
2B	u/mod	(u1 u2 -- urem uquot)
89	unloop	(--) (R: sys --)
TG	until	(C: dest-sys --) (done? --)
81	upc	(char1 -- char2)

Table 96 FCodes by Name *(Continued)*

Value	**Function**	**Stack**
9C	u.r	(u size --)
0219	user-abort	(… --) (R: … --)
TG	value	(E: -- x) (x "new-name< >"--)
TG	variable	(E: -- a-addr) ("new-name< >"--)
FD	version1	(--)
74	w!	(w waddr --)
D1	w,	(w --)
6F	w@	(waddr -- w)
5B	/w	(-- n)
67	/w*	(nu1 -- nu2)
70	<w@	(waddr -- n)
5F	wa+	(addr1 index -- addr2)
63	wa1+	(addr1 -- addr2)
80	wbflip	(w1 -- w2)
0236	wbflips	(waddr len --)
AF	wbsplit	(w -- b1.lo b2.hi)
TG	while	(C: dest-sys -- orig-sys dest-sys) (continue? --)
0166	window-left	(-- border-width)
0165	window-top	(-- border-height)
45	within	(n min max -- min<=n<max?)
7D	wljoin	(w.lo w.hi -- quad)
0221	wpeek	(waddr -- false | w true)
0224	wpoke	(w waddr -- okay?)
0244*	wxjoin	(w.lo w.2 w.3 w.hi -- o)
0245*	x,	(o --)
0246*	x@	(oaddr -- o)
0247*	x!	(o oaddr --)
0248*	/x	(-- n)
0249*	/x*	(nu1 -- nu2)
024A*	xa+	(addr1 index -- addr2)
024B*	xa1+	(addr1 -- addr2)
024C*	xbflip	(oct1 -- oct2)
024D*	xbflips	(oaddr len --)
024E*	xbsplit	(o -- b.lo b.2 b.3 b.4 b.5 b.6 b.7 b.hi)
024F*	xlflip	(oct1 -- oct2)
0250*	xlflips	(oaddr len --)
0251*	xlsplit	(o -- quad.lo quad.hi)
25	xor	(x1 x2 -- x3)
0252*	xwflip	(oct1 -- oct2)
0253*	xwflips	(oaddr len --)
0254*	xwsplit(	(o -- w.lo w.2 w.3 w.hi)

B

Coding Style

This appendix describes the coding style used in some Open Firmware implementations. These guidelines are a “living” document that first came into existence in 1985. By following these guidelines in your own code development, you will produce code that is similar in style to a large body of existing Open Firmware work. This will make your code more easily understood by others within the Open Firmware community.

Typographic Conventions

The following typographic conventions are used in this document:

- The symbol ⁘ is used to represent space characters (i.e. ASCII 0x20).
- The symbol … is used to represent an arbitrary amount of Forth code.
- Within prose descriptions, Forth words are shown in `this` font.

Use of Spaces

Since Forth code can be very terse, the judicious use of spaces can increase the readability of your code.

Two consecutive spaces are used to separate a definition’s name from the beginning of the stack diagram, another two consecutive spaces (or a newline) are used to separate the stack diagram from the word’s definition, and two consecutive spaces (or a newline) separate the last word of a definition from the closing semi-colon. For example:

```
: new-name⁘⁘(⁘stack-before⁘--⁘stack-after⁘)⁘⁘foo⁘bar⁘⁘;
```

```
: new-name⁘⁘(⁘stack-before⁘--⁘stack-after⁘)
⁘⁘⁘foo⁘bar⁘framus⁘dup⁘widget⁘foozle⁘ribbit⁘grindle
;
```

Forth words are usually separated by one space. If a phrase consisting of several words performs some function, that phrase should be separated from other words/phrases by two consecutive spaces or a newline.

```
: name⁘⁘(⁘stack before⁘--⁘stack after⁘)⁘⁘this1⁘this2⁘⁘that1⁘that2⁘⁘;
```

If you are uncertain how to group words into phrases, one useful algorithm is to look at the stack effect of a group of words. A group of words is a phrase when the group has no net stack effect (i.e. the stack looks the same after the group has executed as it did before the group was executed).

When creating multiple line definitions, all lines except the first and last should be indented by three (3) spaces. If additional indentation is needed with control structures, the left margin of each additional level of indentation should start three (3) spaces to the right of the preceding level.

```
: name ( stack before -- stack after )
␣␣␣qqq…
␣␣␣␣␣␣qqq…
␣␣␣␣␣␣qqq…
␣␣␣qqq…
;
```

if…else…then

In `if…then` or `if…else…then` control structures that occupy no more than one line, two spaces should be used both before and after each `if`, `else` or `then`.

```
␣␣if␣␣qqq␣␣then␣␣
␣␣if␣␣qqq␣␣else␣␣ppp␣␣then␣␣
```

Longer constructs should be structured like this:

```
<code to generate flag>␣␣if
␣␣␣<true clause>
then
<code to generate flag>␣␣if
␣␣␣<true clause>
else
␣␣␣<false clause>
then
```

do…loop

In `do…loop` constructs that occupy no more than one line, two spaces should be used both before and after each `do` or `loop`.

```
<code to calculate limits>␣␣do␣␣qqq␣␣loop␣␣
```

Longer constructs should be structured like this:

```
<code to calculate limits>␣␣do
␣␣␣<body>
loop
```

The longer `+loop` construct should be structured like this:

```
<code to calculate limits>␣␣do
␣␣␣<body>
<incremental value>␣+loop
```

begin…while…repeat

In `begin…while…repeat` constructs that occupy no more than one line, two spaces should be used both before and after each `begin`, `while` or `repeat`).

```
␣␣begin␣␣<flag code>␣␣while␣␣<body>␣␣repeat␣␣
```

Longer constructs:

```
begin␣␣<short flag code>␣␣while
␣␣␣<body>
repeat
begin
␣␣␣<long flag code>
while
␣␣␣<body>
repeat
```

begin…until…again

In `begin…until` and `begin…again` constructs that occupy no more than one line, two spaces should be used both before and after each `begin`, `until` or `again`.

```
␣␣begin␣␣<body>␣␣until
␣␣begin␣␣<body>␣␣again
```

Longer constructs:

```
begin
␣␣␣<body>
until
begin
␣␣␣<body>
again
```

Block Comments

Block comments begin with `\␣`. All text following the space is ignored until after the next newline. (While it would be possible to delimit block comments with parentheses, the use of parentheses is reserved by convention for stack comments.

Precede each non-trivial definition with a block comment giving a clear and concise explanation of what the word does. Put more comments at the very beginning of the file to describe external words which could be used from the User Interface.

Stack Comments

Stack comments begin with `(`✥ and end with `)`. Use stack comments liberally within definitions. Try to structure each definition so that, when you put stack comments at the end of each line, the stack picture makes a nice pattern.

```
: name ( stack before -- stack after )
✥✥✥qqq ppp bar ( stack condition after the execution of bar )
✥✥✥qqq ppp foo ( stack condition after the execution of foo )
✥✥✥qqq ppp dup ( stack condition after the execution of dup )
;
```

Return Stack Comments

Return stack comments are also delimited with parentheses. In addition, the notation `r:` is used at the beginning of the return stack comment to differentiate it from a parameter stack comment.

Place return stack comments on any line that contains one or more words that cause the return stack to change. (This limitation is a practical one; it is often difficult to do otherwise due to lack of space.) The words `>r` and `r>` must be paired inside colon definitions and inside `do…loop` constructs.

```
: name ( stack before -- stack after )
✥✥✥qqq >r ( r: addr )
✥✥✥qqq r>( r: )
;
```

Numbers

Hexadecimal numbers should be typed in lower case. If a given number contains more than 4 digits, the number may be broken into groups of four digits with periods. For example:

```
dead.beef
```

Since the default number base is hexadecimal, the convention is not to precede hexadecimal numbers with `h#`.

Optimizations

A number of commonly used operations have been given optimized definitions. The use of these optimizations will improve the performance and reduce the size of your code.

Table 97 Forth Optimizations

Use	Instead of
1+	1 +
1-	1 -
2+	2 +
2-	2 -
2*	2 *
2/	2/
0=	0 =
0<	0 <
0>	0 >
0<=	0 <=
0>=	0 >=
0<>	0 <>

Case Insensitivity

Forthmacs is case insensitive. However, the convention is to use lower case for Forth words. Upper case characters may be used in comments while typing a regular English text.

```
\ This is an example comment
```

Index